Desarrollo de un sistema de análisis del riesgo financiero

Por
Gerardo Parra Rossignoli

UNIVERSIDAD COMPLUTENSE
MADRID

Grado en Ingeniería del Software
Facultad de Informática

Antonio Sarasa Cabezuelo
Desarrollo de un sistema de análisis del riesgo financiero

Madrid, 2020–2021

Development of a financial risk analysis system

By
Gerardo Parra Rossignoli

UNIVERSIDAD COMPLUTENSE
MADRID

Software Engineering Degree
Computer science faculty

Antonio Sarasa Cabezuelo
Development of a financial risk analysis system

Madrid, 2020–2021

Autorización de difusión

El abajo firmante, matriculado en el Grado en Ingeniería del Software de la Facultad de Informática, autoriza a la Universidad Complutense de Madrid (UCM) a difundir y utilizar con fines académicos, no comerciales y mencionando expresamente a su autor, el presente Trabajo Fin de Grado: 'Desarrollo de un sistema de análisis del riesgo financiero', realizado durante el curso académico 2020-2021 bajo la dirección de Antonio Sarasa Cabezuelo, en el Departamento de Sistemas Informáticos y Computación, y a la Biblioteca de la UCM, a depositarlo en el Archivo Institucional E-Prints Complutense con el objeto de incrementar la difusión, uso e impacto del trabajo en Internet y garantizar su preservación y acceso a largo plazo.

Gerardo Parra Rossignoli
12 de Junio de 2021

Dedicatorias

A mis padres, por enseñarme con su ejemplo el valor de la constancia y la dedicación.

A mi familia, por su apoyo incondicional.

A Alicia, por estar siempre ahí, en mis victorias y en mis derrotas.

A mis amigos Alberto, Iván y Adrián por acompañarme durante todo el camino y ayudarme a mejorar como profesional y como persona.

Agradecimientos

Quiero agradecer a todos mis profesores del grado, por poner a mi disposición todo el tiempo y los recursos para obtener la mejor formación. En especial al tutor de este trabajo, Antonio Sarasa Cabezuelo, por su ayuda y guía durante todo el proceso.

Resumen

Desde la última gran recesión económica en el año 2008, el sistema económico y político global, y en especial a nivel Europeo, ha sufrido grandes cambios regulatorios y de políticas fiscales. Parece que hoy en día, esta tendencia va a continuar y cada vez más, se hace ver en el panorama público la necesidad de que los ciudadanos mejoren su cultura financiera y capacidad de generación de ahorro.

Por ello, en este Trabajo de Fin de Grado, se presenta la implementación de un sistema compuesto por diferentes servicios web, que permita hacer accesible algunos cálculos y métricas aplicadas a la gestión del riesgo en la inversión, con el objetivo de que los usuarios puedan probar diferentes situaciones y escenarios, y así aumentar su confianza y preparación financiera.

Palabras clave

Inversión financiera, gestión del riesgo, microservicios, python, django, flask.

Abstract

Since the last major economic recession in the 2008, the global economic and political system, especially in Europe, has undergone significant changes in fiscal policies and regulations. It seems that nowadays this trend will continue, and more and more the need for citizens to improve their financial education and their saving capacity is taking space in the public debate.

Therefore, in this Final Degree Project, it is presented the implementation of a system composed of different web services, which allows to make accessible some calculations and metrics applied to risk management in investment, with the aim that users can test different situations and scenarios, and thus increase their confidence and financial expertise.

Keywords

Financial investment, risk management, microservices, python, django, flask.

Índice general

Capítulo 1

Introducción

1.1. Contexto

En el último siglo, la humanidad ha experimentado un avance vertiginoso en el desarrollo científico y tecnológico que ha llevado a grandes cambios en cómo se entiende y organiza la sociedad.

Se ha experimentado la mejora y optimización de procesos productivos que ha permitido el abaratamiento de costes y el aumento de la oferta de todo tipo de productos. Esto también ha conllevado, por ejemplo, la pérdida de puestos de trabajo y el empobrecimiento del tejido productivo de las sociedades occidentales como consecuencia de la deslocalización de las fábricas, que se han ido trasladando a zonas donde el coste de producción es más barato.

Términos como *Transformación Digital* o *Inteligencia Artificial* se están haciendo hueco en el día a día. Son la representación de la mejora tecnológica y proporcionarán profundos y positivos cambios en el modo de vida de la sociedad. Aunque, de nuevo, ésta se enfrenta a la pérdida de puestos de trabajo que serán automatizados y a la necesidad de mayor cualificación de la población productiva. Además, sin ir más allá de la próxima década, en este año 2020-2021, la humanidad se enfrenta a las consecuencias de la pandemia global de la *COVID-19* [1], que podría desencadenar cambios en el modelo productivo internacional con consecuencias inmediatas en el sector servicios y turístico a nivel mundial.

1.2. Motivación

Partiendo del contexto anterior, se puede afirmar que, para proteger la generación de riqueza y el patrimonio personal es, hoy más que nunca, necesario mejorar la educación financiera y el acceso a los mercados para todo el mundo.

En los últimos años, han proliferado muchas empresas en el sector de las *FinTech*, con la intención de cambiar la forma en la que la población se relaciona con la actividad financiera. Donde antes la única opción para acceder a los mercados financieros era hacerlo a través de bancos y cajas de ahorro, hoy en día, se puede hacer mediante una multitud

de empresas y servicios, por ejemplo con gestores de inversión pasiva como *Finizens* [2].

Estas empresas, sin embargo, no consiguen deshacerse de la sensación de inseguridad que la inversión en bolsa suele generar, debido a que son demasiado técnicas y con muchos servicios, lo que hace que sean complicadas de usar para cualquier usuario. En este sentido, existen multitud de fuentes gratuitas y de pago en internet para aprender a invertir, sin embargo, el público puede verse sobrepasado por tal cantidad de información. Además, la falta de formación puede hacer que éste no sea capaz de filtrar el contenido y acabe siendo víctima de estafas. En especial, en los últimos tiempos en algunas plataformas de entretenimiento abiertas a todos los públicos, ha aumentado el número de anuncios de cursos de inversión que prometen objetivos irreales y cuentan historias poco verosímiles sobre los resultados que se pueden lograr mediante la inversión.

Por todo ello, en este Trabajo de Fin de Grado, se va a implementar una solución tecnológica que permita al público simplificar algunas de las operaciones típicas relacionadas con el análisis del riesgo y resultados de una cartera de inversión.

1.3. Objetivos

Este trabajo de fin de grado tiene como objetivo principal el desarrollo de un sistema que permita al usuario una aproximación al análisis del riesgo de sus estrategias de inversión. El sistema desarrollado deberá proveer al usuario de una herramienta que le permita poner a prueba sus hipótesis a la hora de invertir, dándole la opción de simular diferentes escenarios y proporcionándole datos relevantes de rendimiento y riesgo.

En concreto los objetivos son:

- **Desarrollo de un servicio para recoger datos financieros**: Un servicio web que debe ser capaz de recoger de forma periódica información financiera y procesarla para después propagarla al resto del sistema.
- **Desarrollo de un servicio de cálculos financieros**: Este servicio web debe ser capaz de utilizar los datos financieros disponibles en el sistema para realizar cálculos estadísticos referentes a su rendimiento y riesgo, y ofrecerlos al resto del sistema.
- **Desarrollo de una aplicación web**: Una aplicación web que aprovechando los cálculos disponibles proporcione a los usuarios la capacidad de consultar información de los mercados financieros y probar diferentes escenarios para sus carteras de inversión.
- **Reutilización y escalabilidad**: Mostrar una aproximación a este tipo de sistemas que permita una reutilización y escalabilidad de cada componente. Partiendo de los componentes implementados en este trabajo, se debería poder llevar a cabo la implementación de otros elementos que amplíen las capacidades del sistema en su servicio a los usuarios actuales y futuros.

1.4. Estructura del documento

La memoria estará dividida en los siguiente apartados:

1. **Introducción**: Breve resumen del proyecto, motivaciones y objetivos de éste.
2. **Estado del arte**: Descripción del estado actual de los sistemas relacionados con este proyecto.
3. **Tecnología utilizada**: Explicación de las tecnologías utilizadas para el proyecto.
4. **Arquitectura**: Contiene una explicación de la arquitectura del proyecto, los componentes utilizados y la interacción entre ellos.
5. **Modelo de datos**: Explicación de los datos y cálculos utilizados en el proyecto.
6. **Implementación**: Explicación de los pasos y decisiones tomadas durante el desarrollo.
7. **Conclusiones y Trabajo Futuro**: Conclusiones de los resultados y propuestas de mejora y ampliación.
8. **Anexo**: Se presentan dos anexos. El primero es una guía de uso de la aplicación web. El segundo, una explicación detallada del test de perfil financiero disponible en la aplicación web.

Capítulo 2

Introduction

2.1. Context

During the last century, humanity has experienced an incredible advance in scientific and technological development, which has brought deep changes in the way society is understood and organized.

The improvement and optimization of production processes has made it possible to reduce costs and increase the supply of all kinds of products, but it has also led, for example, to job losses and the impoverishment of the productive fabric of Western societies as a result of the relocation of factories to regions where production costs are lower.

Concepts such as *Digital Transformation* or *Artificial Intelligence* are becoming part of everyday life. They are part of the embodiment of technological improvement and will bring major and positives changes in the way society lives. Although, again, society is facing job losses that will be automated and the need for higher qualification of the productive population.
Moreover, without looking beyond the next decade, this same year 2020-2021, humanity is facing the consequences of the global pandemic of *COVID-19* [1] disease, which may led to changes in the international productive model, with inmediate impact in the turistic and tertiary sectors worldwide.

2.2. Motivation

Starting from the above context, it can be affirmed that, to protect the generation of wealth and personal assets, it is more necessary than ever to improve financial education and public access to international markets.

In recent years, many companies have proliferated in the *FinTech* sector, with the intention of changing the way in which the population relates to financial activity. Where previously the only option for accessing financial markets was to do so through banks, today, it is possible to do so through a multitude of companies and services, for example, with passive investment managers such as *Finizens* [2].

These companies, however, do not manage to eliminate the feeling of insecurity that financial investment usually generates, since they are too technical and offer too many services that make them complex to use for any type of user. In this sense, there are many free and paid sources on the Internet to learn how to invest, nevertheless, the public can be overwhelmed by such a volume of information. In addition, the lack of training may cause the user not to know how to filter the contents and end up being a victim of a scam. In particular, there is a growing trend on some entertainment platforms open to all audiences, of advertisements for investment courses that promise unrealistic goals and tell implausible stories about the returns that can be obtained through investment.

Therefore, in this Final Project a technological solution will be implemented to allow to simplify to the public some typical operations related to the analysis of risk and results of an investment portfolio.

2.3. Objectives

The main objective of this final degree project is the development of a system that allows the user an approach to the risk analysis of his investment strategies. The developed system should provide the user with a tool that enables him to test his hypotheses when investment, giving him the option to simulate several scenarios and providing him with relevant performance and risk data.

Specifically, the objectives are:

- **Development of a service to collect financial data**: A web service that must be able to periodically collect financial information and process it for later propagation to the rest of the system.
- **Development of a financial calculation service**: This web service must be able to use the financial data available in the system to perform statistical calculations regarding their performance and risk, and provide them to the rest of the system.
- **Web application development**: A web application that leverages available calculations to provide users with the ability to query financial market information and test different scenarios for their investment portfolios.
- **Reusability and scalability**: To show an approach to this type of systems that allows a reusability and scalability of each component. Based on the components implemented in this work, it should be possible to carry out the implementation of other elements that expand the capabilities of the system in its service to current and future users.

2.4. Structure of the document

The memory will be divided into the following sections:

1. **Introduction**: Brief summary of the project, motivations and objectives.
2. **State of the art**: Description of the current state of the systems related to this project.
3. **Technology used**: Explanation of the technologies used for the project.
4. **Architecture**: Contains an explanation of the project architecture, the components used and the interaction between them.
5. **Data model**: Explanation of the data and calculations used in the project.
6. **Implementation**: Explanation of the steps and decisions taken during development.
7. **Conclusions and Future Work**: Conclusions of the results and proposals for improvement and expansion.
8. **Appendix**: Two appendices are presented. The first is a guide to the use of the web application. The second is a detailed explanation of the financial profile test available in the web application.

Capítulo 3

Estado del arte

Como ha sido expuesto con anterioridad, en el mercado existen multitud de opciones para realizar operaciones de inversión. A modo de referencia, como se explica en este artículo, *What Are the Different Types of Fintech?* [3], podemos encontrar diferentes tipos de Fintech clasificándolas por su dominio. Este trabajo se centra en los tipos de *trading*, *robo-advising y stock-trading*, y *finanzas personales o wealthTech*.

El tipo *trading*, hace referencia a brokers como *eToro* [4] o *Plus500* [5]. Estas aplicaciones ofrecen un servicio muy completo de ciertas opciones de inversión y proporcionan cuentas demo que permiten simular una evolución real de su inversión al usuario. Su principal desventaja es que exponen al usuario a una interfaz de inversión completa y la curva de aprendizaje requerida puede causar rechazo en él.

Respecto al tipo de *robo-advising y stock-trading*, existen apps como *Finizens* [2] o *Indexa Capital* [6]. Estos productos ofrecen la gestión pasiva de carteras de inversión, es decir, el cliente delega la gestión de su dinero en la empresa. Esta opción de inversión es atractiva y relativamente recomendable, pero no todas las personas confían en dejar su dinero en manos de otro.

Por último en esta clasificación, están las Fintech de tipo *wealthTech*, centradas en la gestión del patrimonio personal, por ejemplo, *Mint* [7]. Esta app permite al usuario organizar sus cuentas, balances y uso de tarjetas de crédito a cuentas de inversión y préstamos, además tiene recordatorios de pagos y permite hacer presupuestos orientativos y objetivos de gastos.

Por otro lado, existen herramientas de visualización del rendimiento de un portfolio, como pueden ser *Portfolio Visualizer* [8] o *Portfolio Performance* [9]. Este proyecto esta estrechamente relacionado con este tipo de herramientas muy completas. Tienen dos desventajas, al igual que las anteriores presentan gran cantidad de información, y además en este caso, no tienen traducción oficial al español.

En las siguientes secciones del capítulo, se presentan los elementos financieros sobre los que este trabajo va a construirse. En la sección 3.1, se introduce la idea de la cartera de acciones y sobre que conceptos clave se construye ésta. Después, en la sección 3.2, se introduce la definición de acciones y en que mercados se pueden operar. Por último, en la sección 3.3, se expone cómo y de dónde se pueden obtener datos financieros.

3.1. La cartera de inversión o Portfolio

Un elemento común entre todas las aplicaciones citadas con anterioridad es la cartera de inversión (en adelante, el Portfolio).

El Portfolio es la colección de inversiones, ya sean acciones, bonos, o cualquier instrumento financiero, que el inversor posee. Tanto si el inversor ha elegido gestionarlo por si mismo, de forma activa, como si prefiere delegar su gestión, de forma pasiva, es importante comprender algunos conceptos para asegurar que la inversión dará la rentabilidad esperada. En este sentido los dos conceptos clave son el rendimiento y el riesgo.

El rendimiento hace referencia a los resultados de las inversiones que componen el Portfolio. El riesgo es la probabilidad de que los resultados de la inversión se conviertan en pérdidas.

Toda inversión tiene riesgos, que varían dependiendo del tipo de activo. Existe una relación [10] entre el riesgo y la rentabilidad de un activo, a mayor rentabilidad, mayor riesgo y viceversa. Así por ejemplo, un bono estatal tendrá un riesgo muy bajo dado que se trata de un instrumento de rentabilidad fija y amparado por un estado. Por otro lado, un derivado de una acción, como los que ofrece cualquier *broker* mencionado con anterioridad en esta categoría, tiene gran rentabilidad pero un enorme riesgo.

Por tanto, es necesario conocer bien cuál es el riesgo que se toma al realizar una operación, y monitorizar correctamente el rendimiento que se consigue para saber cuándo y cómo desechar algunos activos u obtener nuevos. Así, existen dos elementos esenciales para el inversor, la *tolerancia al riesgo* y la *diversificación del Portfolio*.

La tolerancia al riesgo es una forma de medir el desgaste psicológico que un inversor puede aguantar en relación a los riesgos que toma. La Comisión Nacional del Mercado de Valores (CNMV) define la aversión al riesgo en su glosario financiero [11] como: "la actitud de rechazo que experimenta un inversor ante el riesgo financiero, en concreto ante la posibilidad de sufrir pérdidas en el valor de sus activos". Para representar esta tolerancia del inversor, establece tres niveles: *conservador*, *medio* y *agresivo*. Esto tiene implicaciones en la construcción del Portfolio, ya que marcará que activos deberían utilizarse, y sobre todo, cuáles no.

En cuanto a la diversificación, se trata de la estrategia que se debe seguir cuando se construye el Portfolio con el objetivo de minimizar el riesgo de éste. El Portfolio, debería estar formado por activos con distintos grados de correlación y riesgo entre sí. De esta forma si un mercado determinado pasa por un periodo bajista, no todos los activos del Portfolio se verán afectados, dando capacidad de maniobra al inversor.

3.2. Mercados de acciones

Según la CNMV, una acción [12] es: "una parte proporcional del capital social de una sociedad; por tanto, los accionistas son copropietarios de las empresas en proporción a su participación. Son valores participativos negociables y libremente transmisibles."
Existen dos tipos de mercado en los que se vende una acción, el primario y el secundario.

El mercado primario es el mercado de emisión de valores o títulos financieros que son de nueva creación. Este mercado es utilizado por las empresas con el objetivo principal de captar fondos del público mediante la emisión de nuevos valores. Es decir, los inversores obtienen títulos con valor de primera emisión que adquieren directamente del emisor.

En los mercados primarios la emisión de los productos se realiza y se verifica bajo determinados criterios de liquidez y solvencia, así como la valoración de los activos financieros se produce a través de agencias de rating. La emisión se realiza mediante la definición de las características del producto a emitir y su colocación se lleva a cabo a través de agencias emisoras autorizadas que tengan una determinada reputación en el mercado, con la finalidad de que pueda ser negociado en el mercado secundario.

Por tanto, el mercado primario es el primer filtro de mercado y es fundamental para garantizar un buen control del instrumento financiero emitido.

Por otra parte, el mercado secundario es el mercado de negociación o de transacción de operaciones. En él se negocia en función de la oferta, la demanda y otras muchas variables económicas y no económicas, cualquier título financiero tanto de renta fija como de renta variable del mercado monetario.

Por ejemplo, en España existen cuatro bolsas de valores: la Bolsa de Madrid, la de Barcelona, la de Bilbao y la de Valencia. En estas bolsas, la negociación tiene varios segmentos diferentes según el nivel de liquidez de las acciones (tamaño de la empresa y frecuencia de negociación).
El "mercado continuo" es el segmento más líquido de negociación y está basado en un sistema informatizado de contratación en el que se negocian, de forma centralizada, las acciones que cotizan en cualquiera de las cuatro bolsas.

Este mercado es de vital importancia ya que permite que los ahorradores y consumidores canalicen sus intereses, facilitando el intercambio a través de plataformas tecnológicas como las mencionadas anteriormente.

Por último, un índice bursátil es el indicador de un mercado que describe el comportamiento de las cotizaciones de una muestra de valores suficientemente representativa.
Representa la evolución de las empresas de un país, un determinado sector de la economía o un tipo de activo financiero. Gracias a ello, los índices son un excelente indicador de la economía. Por esta razón son utilizados de forma habitual como *benchmark*, es decir, se utiliza el valor del índice como referencia para poder comparar el rendimiento de un Portfolio.

Un ejemplo de índice bursátil español es el IBEX 35. El índice compuesto por los 35

valores más líquidos cotizados de las cuatro Bolsas Españolas, que es usado como referente nacional e internacional.

3.3. Fuentes de Datos

Un requisito para la implementación de la herramienta descrita en este trabajo es el acceso a los datos bursátiles. Para ello ha sido necesario acceder a fuentes de datos financieras online que ofrecen datos bursátiles tal como los precios, el identificador isin, etc.

En general, estas fuentes de datos permiten explotar sus datos a través de APIs de servicios Web de tipo REST o de bibliotecas cliente que encapsulan el acceso a éstos. En la mayoría de los casos se ofrece un acceso limitado a los servicios de manera gratuita o directamente no existe la opción. Su funcionamiento consiste en realizar una consulta a la API y como resultado se devuelve los datos en algún formato como JSON, XML y otros.

Algunos ejemplos de fuentes de datos de tipo bursátil son:

Twelve Data [13]: Ofrece una librería cliente para su API muy cómoda, que proporciona datos de acciones en mercados de todo el mundo en tiempo real con un pequeño retraso de unos 15 minutos.
Sin embargo, esta API en su capa de uso gratuito restringe mucho las llamadas, 8 por minuto con un límite de 800 diarias, y además solo provee datos de acciones estadounidenses.

Alpha Vantage [14]: Es otra API con muchos usuarios, ofrece los datos de acciones de todo el mundo, incluido un histórico de 20 años. En este caso, esta API se expone a través de llamadas HTTP y permite de manera gratuita 5 llamadas por minuto con un límite de 500 llamadas diarias.

Yahoo Finance [15]: Una web en la que se exponen datos en tiempo real de todos los mercados mundiales además de noticias y otros datos relacionados al mundo financiero. Hasta hace unos años, ofrecían una API similar a las anteriormente expuestas, sin embargo, desde 2017 yahoo no presta soporte oficial a esta API, y aunque de manera extraoficial la comunidad ha hecho un mapeo de sus *endpoints*, los datos y llamadas no son estables.
También existen múltiples proyectos de código abierto que permiten *scrapear* los datos directamente de la web, más adelante se entrará en detalle en esta opción.

Capítulo 4

Tecnologías utilizadas

Como se ha expuesto con anterioridad, los objetivos técnicos de este trabajo implican el desarrollo de dos servicios y una aplicación web. En este capítulo se va a explicar las tecnologías empleadas para implementar cada uno de estos servicios.

En la sección 4.1 se expondrán las tecnologías transversales a todo el desarrollo. En la sección 4.2 las tecnologías usadas en el servicio de recolección de datos financieros. En la sección 4.3, las tecnologías aplicadas en el desarrollo del servicio de cálculos. Por último en la sección 4.4, las tecnologías empleadas en la implementación de la aplicación web.

4.1. Tecnologías transversales al desarrollo

4.1.1. Git

Como herramienta de versión de control se ha utilizado *Git* [16]. Esta herramienta es ampliamente utilizada en toda la industria. Es un sistema de control de versiones distribuido de código abierto.

Git establece un sistema de *tres fases o estados* [17]. Los archivos gestionados en un repositorio de Git pueden estar en uno de los tres estados: confirmado, modificado o preparado. Confirmado significa que los datos están almacenados de manera segura en la base de datos local. Modificado, significa que el archivo ha sido modificado y aún no se ha confirmado el cambio. Por último, Preparado significa que un archivo se ha marcado como modificado en su versión actual para incluirlo en la siguiente confirmación.

En relación con los tres estados anteriores, Git gestiona tres secciones por proyecto: el directorio Git, el directorio de trabajo y el área de preparación.
El directorio de Git es donde se almacenan los metadatos y la base de datos de objetos del proyecto. Es la parte más importante de Git, y es lo que se copia al clonar un repositorio desde otra máquina. El directorio de trabajo es una copia de una versión del proyecto, los archivos se sacan de la base de datos comprimida en el directorio de Git y se colocan en disco para su uso o modificación. Por último, el área de preparación es un archivo, generalmente contenido en el directorio de Git, que almacena información de los cambios a añadir en la próxima confirmación.

Esta relación entre los estados y las secciones de Git se muestra de forma gráfica en la figura 4.1

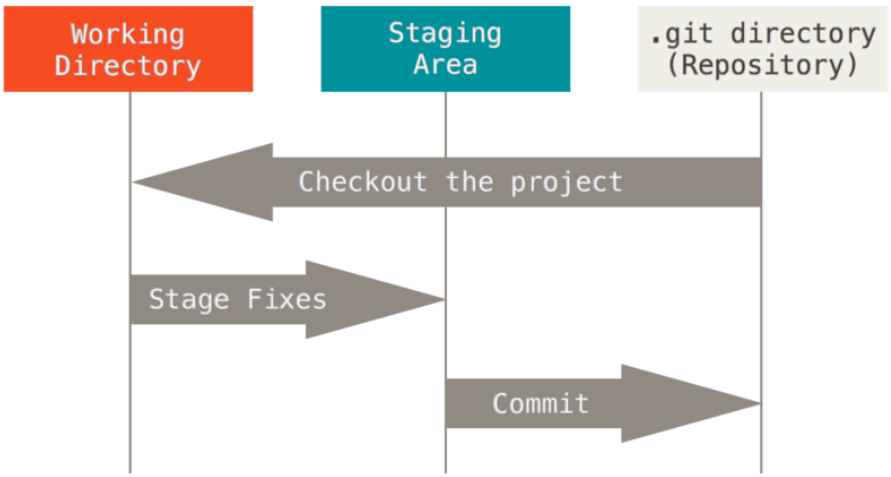


Figura 4.1: Directorio de trabajo, área de almacenamiento y el directorio Git. [17]

Para alojar los distintos repositorios de forma remota, se ha utilizado *Github* [18]. Se trata de una plataforma que ofrece hosting de repositorios Git como servicio principal. Es ampliamente utilizada para mantener y desarrollar proyectos personales, empresariales, o grandes proyectos open-source, como puede ser el propio Git.

4.1.2. Python

El lenguaje de programación utilizado ha sido *Python v.3.9* [19].

Se trata de un lenguaje interpretado multiplataforma. Sus características principales son el uso de un sistema de tipado dinámico y soporte para múltiples paradigmas de programación como son el Orientado a Objetos, el Procedural y el Funcional. Además proporciona una sintaxis clara y relativamente sencilla, dado que uno de sus principales objetivos es la simplicidad y la escritura explícita del código.

4.1.3. Pycharm

Como *IDE* común para todo el trabajo se ha usado *Pycharm Community v.2020.3* [20]. Pycharm es un *IDE* específico para el ecosistema de Python. Algunas de sus características son: una interfaz de edición de código con autocompletado inteligente, resaltado de sintaxis y errores, integración de linter de Python para mantener los *estándares de estilo* [21] y sugerir simplificaciones de estructuras de código. También proporciona una interfaz de *debugging* que permite, además de comprobar el snapshot de la ejecución, probar pequeñas modificaciones de código mientras la ejecución continúa parada.

4.1.4. Virtualenv

Todos los servicios desarrollados tienen dependencias externas. Para evitar que estas dependencias se instalen a nivel de sistema y causen errores e incompatibilidades con otros programas del sistema o entre los propios servicios, se han usado los entornos virtuales que ofrece *Virtualenv* [22]. En Python, los paquetes tanto de la propia versión de Python como de bibliotecas de terceros, se instalan en una carpeta específica por defecto. Para poder desarrollar con múltiples versiones y/o bibliotecas de terceros en el mismo sistema, es necesario utilizar una herramienta de entornos virtuales. De esta forma se permite tener en un sistema múltiples entornos aislados entre sí.

4.1.5. RabbitMQ

Para que el servicio de recolección de datos pueda transmitir la información procesada al resto del sistema de forma desacoplada se ha empleado *RabbitMQ* [23]. Rabbit es un software de colas de mensajería (*message broker*) que implementa el protocolo de mensajería *Advanced Message Queuing Protocol (AMQP)* [24]. Actúa como elemento para desacoplar las comunicaciones entre diferentes servicios, siguiendo un patrón de publicación-subscripción. Como se muestra en la figura 4.2.

Figura 4.2: Diagrama ejemplo de pub-sub con rabbitmq.

4.1.6. Docker y Docker Compose

Para acabar esta sección, queda hablar de *Docker* [25]. Se trata de una plataforma para empaquetar y ejecutar aplicaciones en un entorno parcialmente virtualizado llamado contenedor. Gracias a ello, Docker permite ejecutar múltiples contenedores al tiempo en una sola máquina.

Un *contenedor* [26] es una unidad estándar de software que empaqueta el código y sus dependencias, de tal forma que éste se pueda desplegar de forma rápida y segura en cualquier entorno de computación. La construcción de un contenedor se especifica mediante una *imagen*, que debe incluir todo lo necesario para construir y ejecutar el contenedor, como son: el código, el entorno de ejecución, las dependencias o el propio sistema operativo, entre otros.

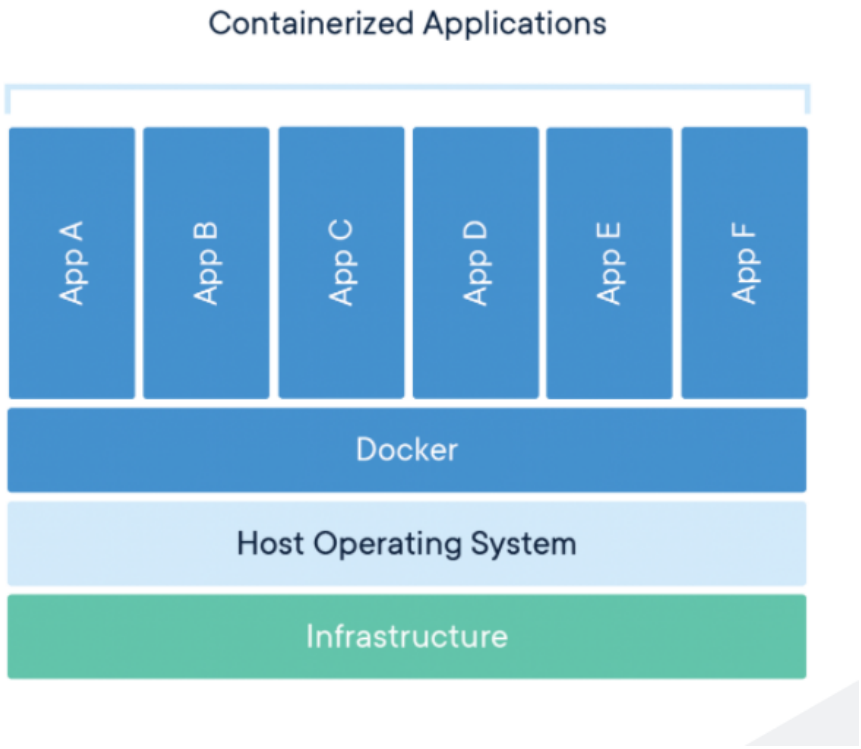


Figura 4.3: Despliegue de varios contenedores en un host. [26]

También se ha utilizado *Docker Compose* [27], una herramienta para definir y ejecutar entornos multicontenedor en un sólo host. Compose permite configurar cada entorno mediante un fichero YAML [28].

4.2. Tecnologías del servicio de recolección de datos

El servicio de recolección de datos utiliza, además de las ya mencionadas en la sección 4.1, bibliotecas específicas para la recolección de los datos.

4.2.1. BeautifulSoup

En primer lugar, para obtener algunos de los datos referentes a los mercados financieros, utiliza técnicas de *web scraping*. El scraping consiste en descargar el código HTML de la página y recorrer su árbol del DOM para obtener información específica.

Para aplicar estas técnicas, el servicio utiliza una biblioteca de Python llamada *BeautifulSoup* [29]. Esta biblioteca provee de una interfaz sencilla para las operaciones de navegación, búsqueda y modificación de los datos recogidos.

4.2.2. MongoDB

Para persistir los datos que son necesarios reutilizar en otros procesos de recolección se ha optado por usar *MongoDB* [30]. Mongo es una base de datos no relacional (NoSQL)

orientada a documentos. Esto significa que almacena los datos en forma de documentos tipo BSON, una representación binaria de JSON, organizados en colecciones. La razón por la que se ha optado por usar esta base de datos y no otra es porque los datos recogidos no tienen ninguna relación entre sí y además carecen de un esquema "fuerte". Por tanto, no son necesarias capacidades propias de una base de datos relacional como pueden ser las operaciones *join*. En concreto, se ha escogido MongoDB porque proporciona soporte ACID [31] completo y por su interfaz sencilla de consultas, en este caso proporcionada a través de la biblioteca cliente *Pymongo* [32].

4.2.3. RabbitMQ

Para enviar los datos recolectados al resto del sistema a través de RabbitMQ, se ha utilizado una implementación que aprovecha la API de *Pika* [33], la biblioteca cliente de RabbitMQ en Python. Pika proporciona una implementación del protocolo *Advanced Message Queuing Protocol (AMQP)* [24] y una API de conexión a RabbitMQ.

4.2.4. APScheduler

Por último, para implementar las ejecuciones de los casos de uso de recolección de forma periódica se ha usado *APScheduler* [34]. Esta biblioteca nos proporciona la capacidad de programar ejecuciones periódicas, mediante el establecimiento de reglas de tipo CRON, basadas en intervalos o de ejecución única.

4.3. Tecnologías del servicio de cálculos financieros

En cuanto al servicio de cálculos financieros, además de las tecnologías propias que se comentarán a continuación, utiliza algunas de las mencionadas en la sección 4.2.

Estas tecnologías comunes son MongoDB y Pymongo, que se han escogido por las mismas razones expuestas en la sección previa. También están RabbitMQ y Pika, que en este caso se implementan para que el servicio pueda consumir los datos enviados por el servicio de recolección.

4.3.1. Pandas y Numpy

Como tecnologías específicas de este servicio aplicadas al cálculo y manejo de los datos se ha usado, *Pandas* [35] y *Numpy* [36]. Pandas es una biblioteca especializada en el análisis de datos orientado a tablas. Esto lo consigue principalmente mediante el uso de los *DataFrame*, estructuras de datos que permiten la manipulación de los datos con indexación por fila y columna. Por otro lado, Numpy es una biblioteca con un catálogo muy amplio de cálculos numéricos. Su estructura de datos básica son los arrays multidimensionales, sobre los que aplica los cálculos. En aplicaciones de naturaleza financiera es muy común encontrar estas dos bibliotecas trabajando juntas.

4.3.2. Flask

Por otra parte, para proporcionar acceso a los cálculos y datos a través de una API de tipo REST se ha utilizado el framework web *Flask* [37].

Flask se define a sí mismo como un "micro-framework", esto significa que a diferencia de otros frameworks que proporcionan al usuario un paquete "pesado" con una gran cantidad de funcionalidad, en flask por defecto sólo se entregan unas funcionalidades básicas, como puede ser: el soporte de servidor HTTP, la gestión de las sesiones, el sistema de plantillas html, etc. Si el usuario desea ampliar estas capacidades debe instalar paquetes adicionales como si fueran una dependencia más, por ejemplo la validación de formularios.

La API de este servicio no necesita capacidades más allá de la gestión del servidor HTTP, por ello, Flask es la opción perfecta.

Para validar las peticiones que llegan a la API, se ha optado por usar *Cerberus* [38], una biblioteca que provee funcionalidades de validación muy configurables y fiables mediante un *schema* de tipo JSON. De esta forma la validación de las peticiones es fácilmente legible y comprensible además de robusta.

4.3.3. Swagger

Para documentar la API, se ha utilizado el editor online de Swagger. Este editor permite difinir un fichero YAML que sigue el estándar *OpenAPI* [39]. Este estándar es agnóstico del lenguaje de programación usado en la API y permite mostrar al cliente los parámetros esperados, las reglas sobre éstos y ejemplos de petición y respuesta.

4.3.4. Postman

Por último, para poder probar las peticiones a la API, se ha elegido *Postman* [40]. Una herramienta que permite preparar peticiones y recibir sus respuestas de una forma cómoda y persistente, organizando estas peticiones en colecciones.

4.4. Tecnologías de la aplicación web

4.4.1. Django

La aplicación web se ha implementado utilizando *Django* [41]. Un framework muy completo y robusto, que proporciona una implementación de alto nivel de un servidor web, desde la gestión de las operaciones HTTP hasta el sistema de autenticación y seguridad de la aplicación.

Django utiliza una arquitectura Modelo-Vista-Plantilla. La vista cumple las funciones de gestión de peticiones y respuestas HTTP. Cada vista responde a una URI concreta y tras validar los datos de entrada, utiliza el Modelo o Modelos necesarios para el caso de uso. Finalmente, renderiza la vista HTML correspondiente con los datos de salida.

Un modelo es un objeto que encapsula la lógica de negocio y la lógica de acceso a datos, siendo ésta gestionada por la implementación de Django. El modelo está estrechamente relacionado con el patrón *Objeto de Negocio*, descrito en el libro *Core J2EE patterns* [42].

Por último, una plantilla es un fichero extensible, que puede implementar partes de otras plantillas, que define un código HTML con marcadores de posición específicos de Django. Estos marcadores se utilizan para representar el contenido real. De esta forma, una vista puede renderizar dinámicamente una página, usando una plantilla que se rellena con datos de un modelo. Finalmente, esta plantilla se devuelve como respuesta HTTP a través de la vista y es renderizada en el navegador del cliente.

En la figura 4.4 se muestra el flujo de trabajo habitual de una aplicación basada en Django.

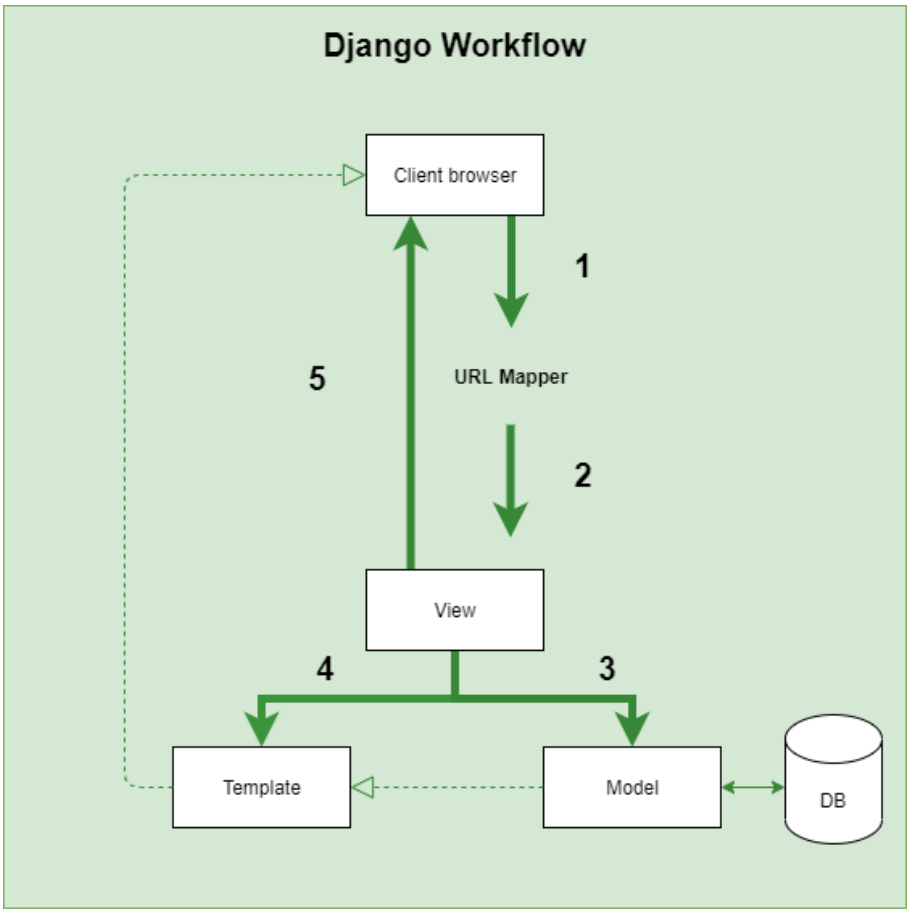


Figura 4.4: Django Modelo-Vista-Plantilla.

En el plano de la persistencia de datos, Django tiene soporte para varias bases de datos. En esta aplicación se ha elegido usar MySQL para almacenar tanto los datos relacionados con la administración de usuarios como puede ser la propia información de cada usuario.

En cuanto a la administración de la web, Django ofrece una interfaz completa que mediante el uso de superusuarios permite gestionar todos los recursos de la web como administrador del sistema.

4.4.2. DBeaver

Para la gestión de la base de datos de la aplicación, se ha utilizado *DBeaver* [43]. Una herramienta de gestión de bases de datos de código abierto. Sus principales ventajas son la facilidad de uso de su interfaz, un entorno integrado de edición y ejecución de SQL y la abstracción que otorga sobre la gestión de dependencias.

4.4.3. Bootstrap

Para desarrollar la interfaz gráfica de la aplicación se ha usado, además de HTML y CSS propios, *Bootstrap* [44]. Un framework de desarrollo Front-End que proporciona una serie de clases HTML con un CSS ya definido con un diseño elegante y funcionalidades de JavaScript integradas.

4.4.4. Chart.js

Para finalizar, en cuanto a la creación de las gráficas de los datos financieros que se pueden visualizar en la aplicación web, se ha elegido usar *Chart.js* [45], una biblioteca implementada con JavaScript para visualizar gráficas interactivas.

Capítulo 5

Arquitectura

En este capítulo se van a explicar las decisiones de arquitectura tomadas para dar respuesta a los objetivos planteados, en especial el referido a la reutilización y escalabilidad de los elementos del sistema.

Para ello, se expondrá primero en la sección 5.1 la arquitectura a alto nivel del sistema. Después, en la sección 5.2 se explicará el diseño del servicio de recolección de datos financieros. En la sección 5.3, el relacionado con el servicio de cálculos. Finalmente en la sección 5.4, el diseño específico de la aplicación web.

Hoy en día, en general, los sistemas empresariales constan de una o varias aplicaciones cliente que proporcionan interfaces de usuario, como una aplicación web Javascript y una aplicación móvil Android. Una aplicación del lado de servidor donde encapsular la gestión de las peticiones y respuestas a los clientes, así como ejecutar la lógica de negocio. Y por último, una base de datos u otra fuente de datos en la que persistir y obtener la información necesaria.

Se puede considerar que existen dos aproximaciones principales a la arquitectura de un sistema.
Por un lado están las arquitecturas monolíticas en las que el sistema, exceptuando las aplicaciones de interfaz, se desarrolla y despliega como una sola unidad. En este tipo de arquitectura, si se lleva a cabo un cambio en una parte de la lógica de negocio, es necesario redesplegar todo el sistema, dado que sus componentes no son independientes, es de esta idea de unidad de la que nace el concepto de monolito. Este tipo de arquitectura tiende con el paso de las iteraciones en su desarrollo a aumentar el acoplamiento de sus componentes, entorpeciendo su escalabilidad vertical. Además de que complica en gran medida la escalabilidad horizontal debido a su "volumen" y cantidad de dependencias.

También plantean un problema desde el punto de vista organizativo para empresas de cierta envergadura. En estas empresas existen múltiples necesidades de negocio que deben ser cubiertas mediante cambios y ampliaciones constantes del sistema. Por ello, las aplicaciones monolíticas son un freno a estas organizaciones dado que los despliegues deben ser pactados entre toda la organización.

Como respuesta a estas desventajas en el lado del servidor, surgen las arquitecturas de microservicios. Esta aproximación pasa por desarrollar la aplicación como un conjunto

de servicios independientes, en los que cada uno lleva a cabo sus procesos de forma desacoplada y altamente cohesiva. La comunicación de estos servicios pasa a ser a través de API o mecanismos asíncronos como puede ser un broker de mensajería.

De esta forma se logra que el sistema gane flexibilidad y agilidad para responder a las necesidades del negocio. Permitiendo así mayor escalabilidad horizontal y vertical, el uso de diferentes tecnologías en cada servicio y ayudando a la organización a poder dividirse en equipos con una lógica mejor enfocada al crecimiento del producto. Disminuyendo por tanto la coordinación necesaria para hacer nuevos despliegues al mínimo necesario.

Sin embargo, este tipo de arquitectura también conlleva problemas típicos de sistemas distribuidos, como pueden ser la gestión de la transaccionalidad distribuida o la necesidad de establecer correctamente los límites funcionales y de dominio de cada servicio.

5.1. Arquitectura del sistema

En este trabajo el sistema requiere dar solución a diferentes objetivos, como son la obtención y tratamiento de datos de diferentes orígenes o el desarrollo de cálculos financieros. Además, es necesario que cada componente sea reutilizable y ampliable en el futuro si se quisiera proveer de nuevas capacidades o modificar las existentes. Por ello se ha decidido aplicar una aproximación de microservicios al sistema.

De esta forma el sistema queda planeado de la siguiente forma:

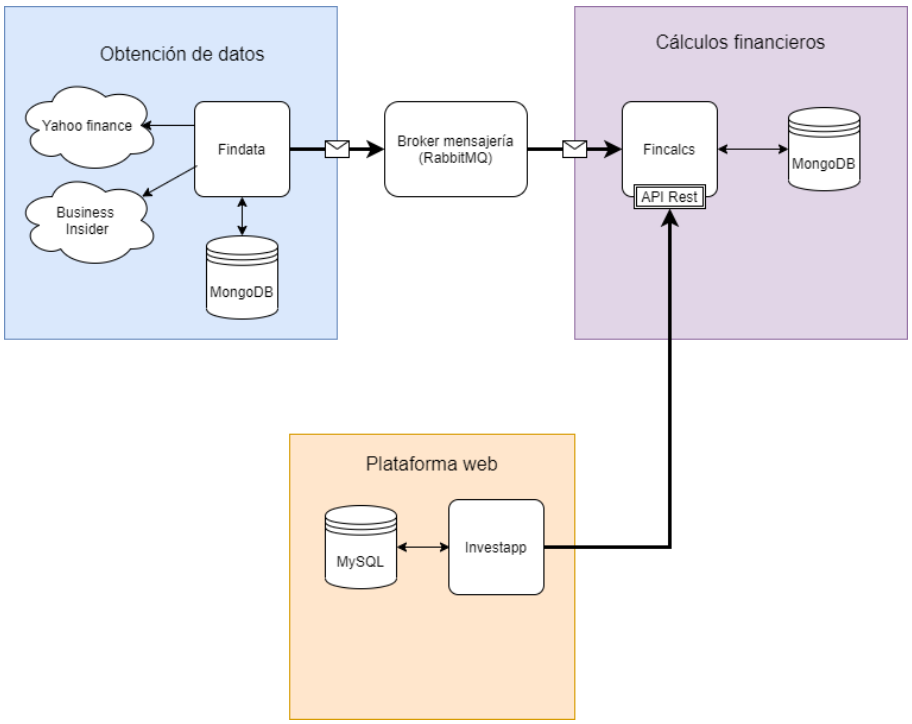


Figura 5.1: Diagrama del sistema.

En la figura 5.1 se muestra el sistema dividido por microservicios. Se puede observar como la comunicación entre los dominios de obtención de datos y de cálculos financieros se produce de forma asíncrona a través de RabbitMQ, implementando así el patrón

Pub-Sub [46]. Por otro lado, entre el dominio de la plataforma web y el de los cálculos, la comunicación es síncrona mediante API Rest.

Por último, durante el desarrollo de microservicios, un impedimento importante es comprobar que el servicio integra correctamente con el resto del sistema. Para mitigar esto existen dos elementos importantes, la gestión del contrato del servicio y la capacidad de disponer de un entorno de prueba con mínima complejidad.

El contrato del servicio es un concepto que suele utilizarse en los productos que proveen su servicio mediante API. Se refiere a que el servicio debe mantener estable su API, de manera que no se produzcan cambios inesperados en sus clientes. Y que todos los cambios, ya sean correcciones de errores o aumentos de funcionalidad deben ser correctamente notificados. Para ello, una herramienta es el *versionado semántico* [47]. Un sistema que establece que cada versión del software debe identificarse por tres números, *x.y.z* (Mayor.Menor.Parche). Cuando se produce la corrección de un error se aumenta el número de parche. Si se aumenta la funcionalidad, sin afectar a la ya existente, se aumenta el número menor. Por último, si se aumenta o modifica la funcionalidad rompiendo la compatibilidad con la versión actual, se aumenta el número mayor.

En cuanto a disponer de un entorno de prueba, Docker es una herramienta perfecta para este caso de uso, como se ha explicado previamente en la sección 4.1, Docker Compose permite montar entornos multicontenedor y ejecutarlos al tiempo. De esta forma se puede disponer de todo el sistema en el entorno de desarrollo local y comprobar la integración de los cambios.

5.2. Servicio de recolección de datos

Este microservicio se encarga de recolectar datos financieros, transformarlos y transmitirlos al resto del sistema. Para ello, debe ser capaz de añadir diferentes fuentes de datos y sobre las ya existentes, aumentar, cambiar o eliminar los datos que recoge. Cualquiera de estos cambios debe afectar el mínimo posible al resto del servicio. Además debe ser capaz de variar de igual modo tanto sus elementos de persistencia de datos como sus medios de transmisión de éstos, por ejemplo, añadiendo un acceso a través de API Rest.

De los anteriores requisitos no funcionales, se puede extraer que el servicio tiene una serie de operaciones relacionadas con el tratamiento de la información y, por otra parte, operaciones ligadas a la propia tecnología como es la obtención de los datos o su persistencia. Por ello, se ha elegido un diseño cercano a la arquitectura hexagonal.

La arquitectura hexagonal tiene como principal objetivo desacoplar la lógica de negocio de la tecnología usada. Tiene tres conceptos que definen los elementos que van a formar el diseño: las entidades, los puertos y los adaptadores.

Las entidades son las que poseen la responsabilidad sobre las reglas del negocio, representan el dominio de la aplicación. Los puertos son interfaces que desacoplan las entidades del resto de la aplicación, son una barrera para evitar que el dominio posea información de una implementación concreta. Por último, los adaptadores son las implementaciones

concretas de los puertos.

Se puede clasificar los puertos en dos tipos, los directores y los dirigidos. Los primeros son aquellos que inician y orquestan la ejecución de los casos de uso y los dirigidos son aquellos que se encargan del acceso a la capa de persistencia o a servicios de terceros.

De esta forma, el diseño del servicio queda así:

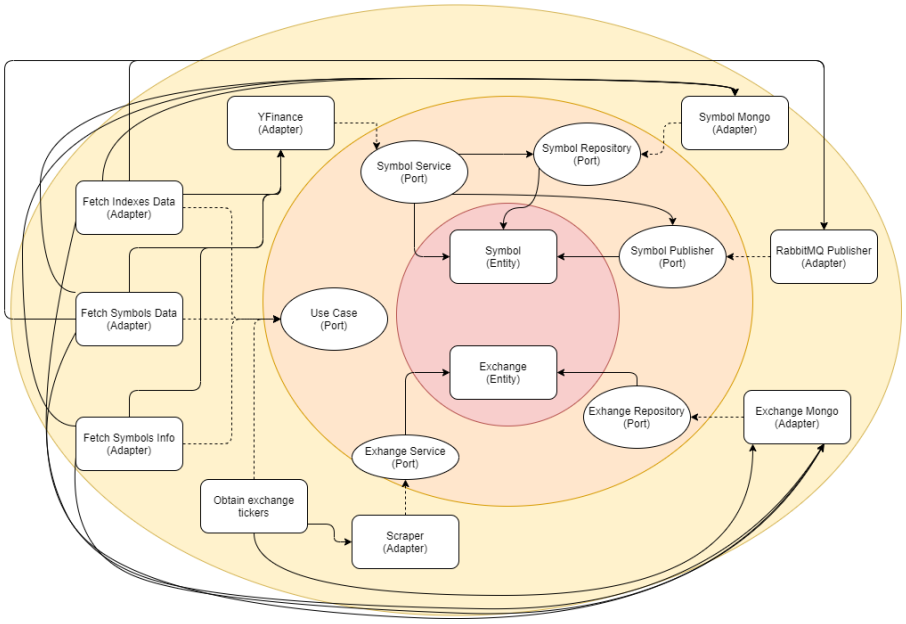


Figura 5.2: Diagrama del diseño del servicio de recolección de datos.

Como se muestra en la figura 5.2, uno de los resultados principales de esta arquitectura es que las dependencias apuntan siempre ”desde fuera hacia dentro”. Es decir, que el dominio queda correctamente desacoplado de las implementaciones concretas.

5.3. Servicio de cálculos financieros

Este microservicio tiene el encargo de persistir los datos financieros y realizar los cálculos estadísticos relacionados con estos datos. Una vez disponible toda esta información, debe ser capaz de ofrecerla al resto del sistema a través de una API Rest. Por ello, al igual que pasaba en la sección 5.2 con el servicio de recolección de datos, existen unas operaciones y funcionalidad que constituyen la lógica de negocio, y por otro lado, las funcionalidades relacionadas con la captación, consumo y exposición de esta información. Por esto, también se ha elegido usar un diseño hexagonal.

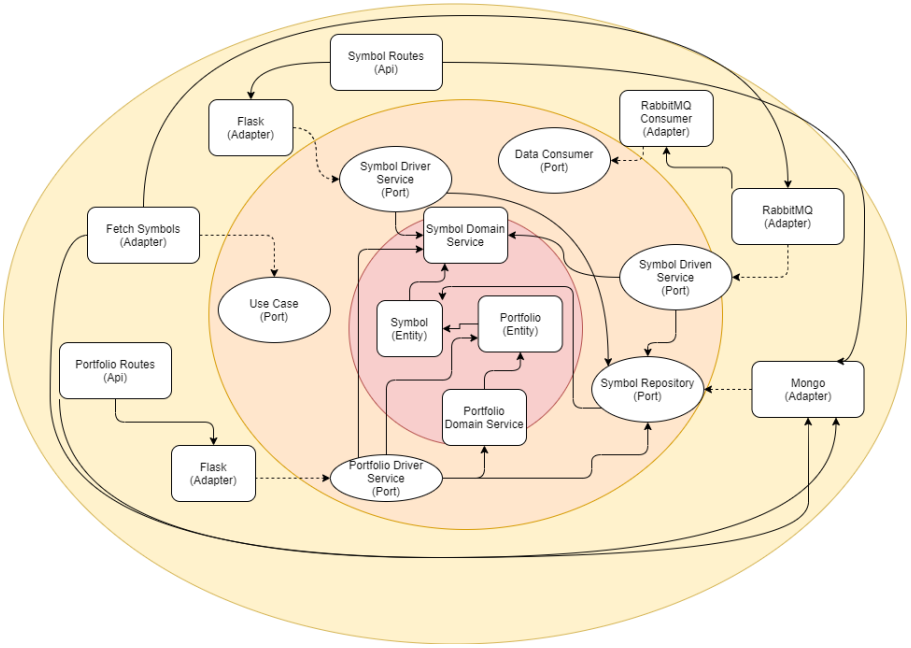


Figura 5.3: Diagrama del diseño de Fincalcs.

En el servicio, hay dos entidades: el símbolo financiero (Symbol) y la cartera de inversión (Portfolio). Sin embargo, no ocurre como en el servicio de recolección de datos, donde sus entidades eran agnósticas entre sí. Aquí el Portfolio está formado por varios Symbols. Por ello, como se puede ver en la figura 5.3, el puerto del Portfolio tiene dependencia con el servicio del dominio de Symbol.

Otro punto a comentar, que no existe en el microservicio de recolección de datos, es el servicio del dominio. Este elemento surge cuando es necesario mantener una lógica creacional y lógica de negocio que afectan a varias entidades o que no encajan en la propia entidad.

Por último, cabe resaltar *Symbol Routes* y *Portfolio Routes*. Estas clases son las encargadas de mapear las URL's de la API y sirven como punto de entrada a los casos de uso correspondientes.

5.4. Aplicación web

La aplicación web ofrece a los usuarios una plataforma para el análisis del riesgo y rendimiento de las acciones y portfolios.

Como se expuso en la sección 4.4, esta implementada con Django. Por tanto, la aplicación sigue un patrón Modelo-Vista-Plantilla (MVT). Al construir una aplicación en Django, se crean dos tipos de artefacto, el primero es el proyecto, que constituye la base y es en el que se crea el fichero *manage.py*, y el *settings.py*.

El fichero *manage.py* es el punto de entrada de Django, y a través de él se realiza la gestión de la base de datos, la creación de superusuarios y el despliegue del servidor. Para gestionar la base de datos, Django mantiene un fichero en el que registra todos los

cambios que se producen en los modelos y en los datos almacenados, a estos registros se les denomina *migrations*. Los superusuarios son los usuarios que obtienen los permisos de administrador de Django. En cuanto al servidor, Django proporciona un servidor de desarrollo, que será el usado en este trabajo, y una interfaz *WSGI* [48] para poder lanzarse en un servidor de producción.

Por otro lado, el fichero *settings.py* es el fichero en el que se configura la aplicación, estableciendo en él las bases de datos, las aplicaciones instaladas, etc.

Además del proyecto, el segundo tipo de artefacto que Django permite crear son las *aplicaciones*. Se puede entender estos artefactos como subproyectos reutilizables entre proyectos. Cada uno tendrá sus propias vistas, modelos y plantillas.

De esta forma la distribución de la aplicación web se divide en el proyecto base, una aplicación para la gestión de perfiles de usuario y otra para la gestión de la información financiera.

Generalmente, en las aplicaciones de Django el punto de entrada se encuentra en la Vista, que instancia al Modelo y posteriormente renderiza la Plantilla correspondiente con la información y la devuelve al navegador como respuesta HTTP.

En la aplicación web, se ha distribuido estos elementos en los paquetes Presentación y Negocio, e Infraestructura en el caso de la aplicación de mercados.

Cabe mencionar también que en ocasiones existe lógica de negocio que no encaja en un Modelo. En los perfiles de usuario el test de perfil de riesgo puede realizarse sin tener sesión activa de usuario, y por tanto en ese caso no existe Modelo correspondiente. En el caso de los mercados, no existen modelos asociados porque no se persiste información, al obtenerse ésta del servicio de cálculos. Para resolver estas situaciones se ha creado una clase semejante a un *Servicio de aplicación* [49], que encapsula la lógica y hace de intermediaria entre la vista y el modelo o el acceso a datos según corresponda.

Por último, para obtener los datos del servicio de cálculos, se ha creado un *DAO* [50] que gestiona la conexión con la API.

Capítulo 6

Modelo de datos

En este capítulo se va a comentar los modelos de datos que utilizan los diferentes servicios de este trabajo. En la sección 6.1 se hablará del modelo de datos específico del servicio de recolección de datos. En la sección 6.2 se expondrá el modelo de datos específico del servicio de recolección de datos. Por último, en la sección 6.3 se explicará el modelo de datos perteneciente a la aplicación web.

6.1. Servicio de recolección de datos

El servicio de recolección de datos utiliza MongoDB para persistir la información, pues como se ha explicado previamente en la sección 4.2, estos datos encajan en un modelo NoSQL.

Como se expuso en la sección 5.2 del capítulo de arquitectura, este servicio tiene dos entidades. La entidad *Exchange* representa un mercado de acciones. Su modelo de datos consiste en un ticker como identificador, una lista de tickers de los Symbols que componen el mercado y la última fecha de modificación.

El ticker es el código específico y único, que un mercado concreto utiliza para identificar esa acción. En este caso el ticker identifica al propio mercado.

En la figura 6.1 se puede ver el Schema de los datos en formato JSON para la entidad de Exchange, con una muestra de los datos recogidos el día 5 de mayo para el IBEX35.

Figura 6.1: Ejemplo JSON del modelo de datos de Exchange.

Para la entidad de Symbol, que representa la información de un símbolo financiero. El modelo de datos correspondiente consiste en el ticker como identificador, el nombre y la última fecha de modificación. Esos tres datos representan un Symbol genérico como puede ser un índice bursátil. Además de esos datos, para un Symbol que pertenezca a una acción, se añaden dos datos adicionales.

Primero el ISIN (International Securities Identification Numbering system) [51], un código que permite identificar a una acción de manera internacional, dado que si cotiza en diferentes mercados, tendrá un ticker distinto en cada uno. Por último, se añade el ticker del exchange al que pertenece. Este dato se añade para facilitar la clasificación de las acciones en otras partes del sistema.

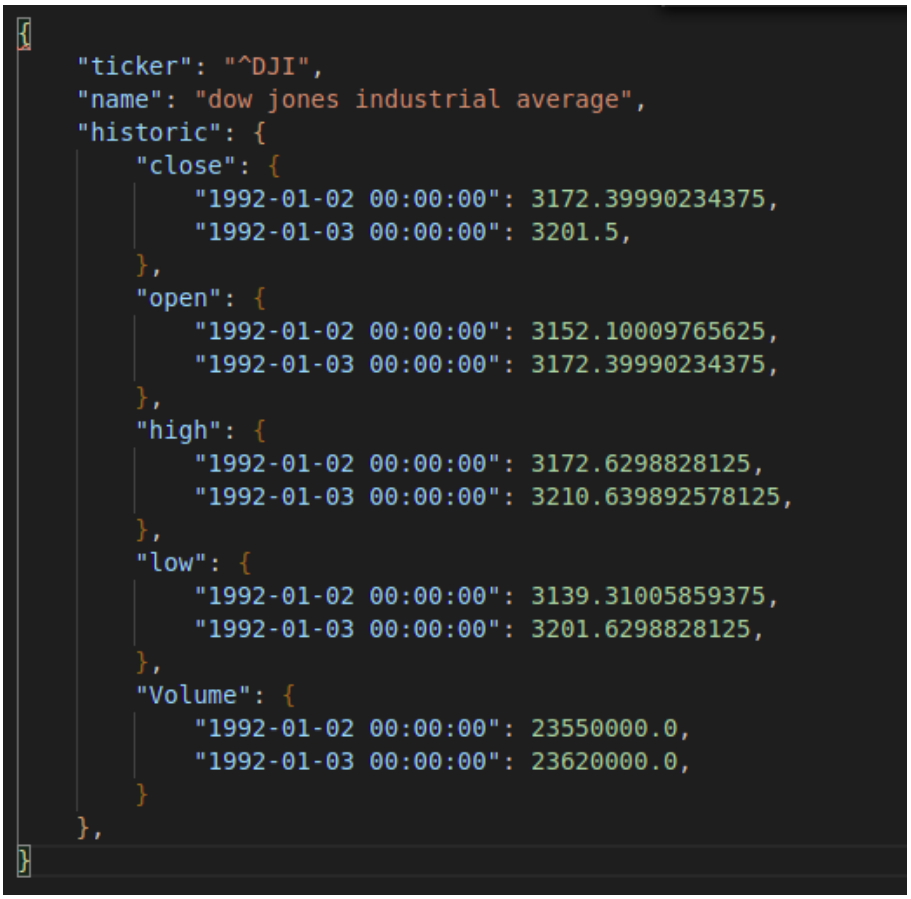
En la figura 6.2 se muestra un ejemplo de Schema en formato JSON para la acción de la empresa española *Acciona s.a.*

Figura 6.2: Ejemplo JSON del modelo de datos de Symbol.

Por otra parte, queda mencionar el modelo de datos de los mensajes que se envían a través de RabbitMQ. En este caso solo se envía información asociada a la entidad de Symbol. La información que se obtiene y envía para todos los símbolos es: el ticker, el

nombre y el histórico, compuesto de los datos diarios de precio de apertura, el máximo y el mínimo y el volumen de la acción. Además del cierre ajustado, este ajuste es el resultado de tener en cuenta eventos corporativos como dividendos o división de acciones. De esta forma, el cierre ajustado provee más contexto. Por último también se añade información de dividendos para los Symbol de tipo Acción.

En la figura 6.3, se muestra un ejemplo de mensaje para un Symbol de tipo Indice, mientra que en la figura 6.4 se muestra un mensaje para un Symbol de tipo Stock.



```
{
    "ticker": "^DJI",
    "name": "dow jones industrial average",
    "historic": {
        "close": {
            "1992-01-02 00:00:00": 3172.39990234375,
            "1992-01-03 00:00:00": 3201.5,
        },
        "open": {
            "1992-01-02 00:00:00": 3152.10009765625,
            "1992-01-03 00:00:00": 3172.39990234375,
        },
        "high": {
            "1992-01-02 00:00:00": 3172.6298828125,
            "1992-01-03 00:00:00": 3210.639892578125,
        },
        "low": {
            "1992-01-02 00:00:00": 3139.31005859375,
            "1992-01-03 00:00:00": 3201.6298828125,
        },
        "Volume": {
            "1992-01-02 00:00:00": 23550000.0,
            "1992-01-03 00:00:00": 23620000.0,
        }
    },
}
```


Figura 6.3: Ejemplo de mensaje de Symbol de tipo Índice.

Figura 6.4: Ejemplo de mensaje de Symbol de tipo Stock.

6.2. Servicio de cálculos financieros

Este servicio, además de usar parte de la información que le llega de RabbitMQ, calcula los retornos diarios del Symbol y los persiste en MongoDB junto al resto de la información. Como identificador se guarda el ticker, closures se refiere a los cierres ajustados, date a la última modificación, name al nombre y además para facilitar su tratamiento por parte del servicio, se especifica su tipo, Stock para acción e Index para un Indice.

Los datos específicos de una Acción son el ISIN, su mercado (Exchange) y los dividendos.

De esta forma, en la figura 6.5 se muestra un ejemplo de Symbol de tipo índice mientras que en la 6.6 uno de tipo acción.

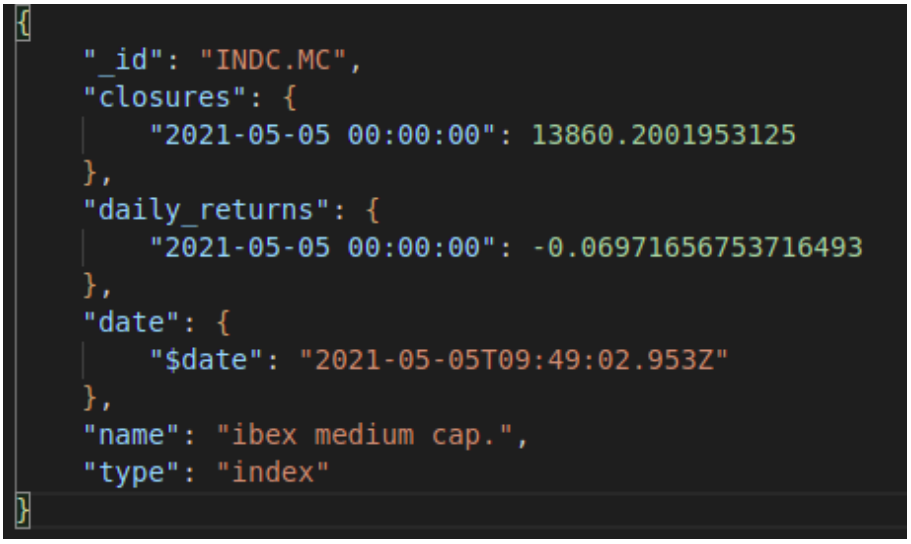


Figura 6.5: Ejemplo de schema en formato JSON de Symbol de tipo Índice.

```
{
    "_id": "NHH.MC",
    "closures": {
        "2021-05-05 00:00:00": 3.9649999141693115
    },
    "daily_returns": {
        "2021-05-05 00:00:00": 0.03255208406092902
    },
    "date": {
        "$date": "2021-05-05T09:49:58.468Z"
    },
    "dividends": {
        "2021-05-05 00:00:00": 0.0
    },
    "exchange": "INDC.MC",
    "isin": "ES0161560018",
    "name": "nh hotel group, s.a.",
    "type": "stock"
}
```


Figura 6.6: Ejemplo de schema en formato JSON de Symbol de tipo Stock.

6.3. Aplicación web

En la aplicación web se manejan dos modelos de datos distintos, por una parte están los datos asociados al perfil de usuario. Por otra, los datos financieros de acciones y mercados.

En este apartado solo se expondrá el modelo de datos del usuario dado que los datos financieros se obtienen a través de la API del servicio de cálculos, y por tanto ya han sido explicados en la sección 6.2.

El perfil de usuario está compuesto por el nombre de usuario, su dirección de correo electrónico y contraseña. Esos datos servirán para la creación del perfil. Posteriormente el usuario puede personalizar su perfil añadiendo el perfil financiero y su índice bursátil preferido.

En la figura 6.7 se puede ver el diagrama Entidad-Relación de la aplicación. Las tablas y campos no mencionados en el párrafo anterior son creados automáticamente por Django. Permiten la gestión de grupos y privilegio desde la interfaz de administrador.

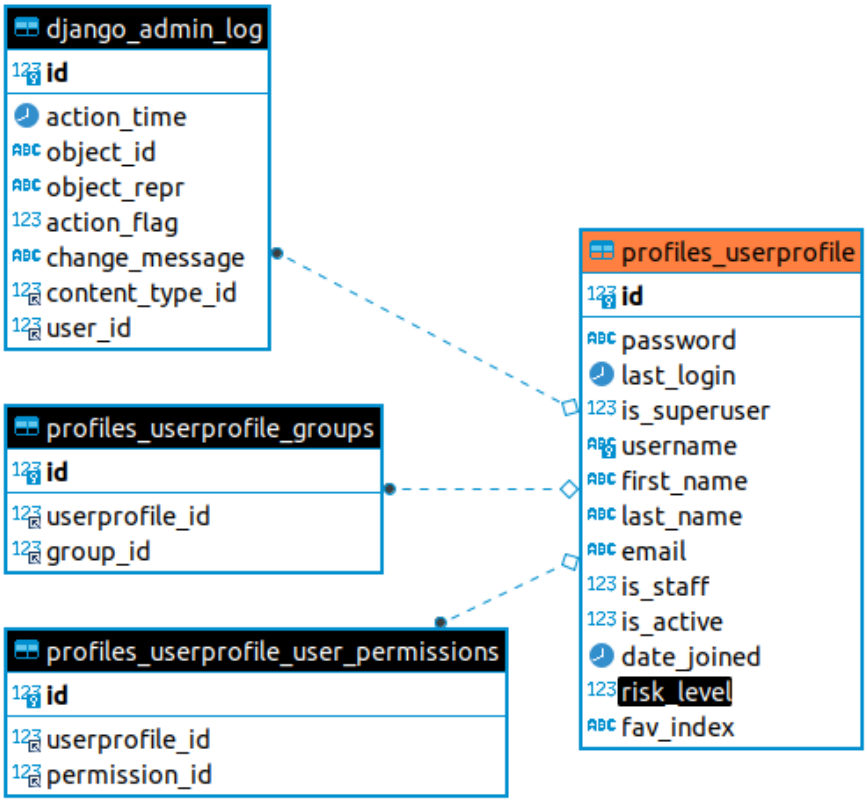


Figura 6.7: Diagrama Entidad-Relación de la BD de la aplicación web.

Capítulo 7

Implementación

En este capítulo se va a explicar las implementaciones concretas de cada uno de los microservicios introducidos en el capítulo 5.1 de arquitectura. El capítulo comenzará por la sección 7.1 en la que se explicarán elementos comunes o muy similares entre microservicios. A continuación, se va a dividir en las secciones 7.2 en la que se hablará del servicio de recolección de datos, la sección 7.3 en la que se explicará la del servicio de cálculos financieros y la sección 7.4 para exponer la implementación de la aplicación web. Para terminar, se explicará la implementación de los contenedores docker en la sección 7.5

Los repositorios que almacenan las implementaciones de los servicios son:

- Servicio de recolección de datos: https://github.com/gprossignoli/findata
- Fork propio de la librería YFinance: https://github.com/gprossignoli/yfinance
- Servicio de cálculos: https://github.com/gprossignoli/FinCalcs
- Aplicación web: https://github.com/gprossignoli/investApp

7.1. Cuestiones generales a todos los microservicios

Respecto a RabbitMQ queda explicar brevemente la implementación seguida en el proyecto. En RabbitMQ los mensajes se envían a un *Exchange* [52], que es el encargado de publicarlo a una de las colas que estén conectadas a él, en función de las reglas que establezca su tipo.

Existen tres tipos de Exchange: el Fanout, el Direct y el Topic. El Fanout emite el mensaje a todas las colas conectadas a él. El Direct se vale de la routing_key, una clave que debe coincidir entre el mensaje y la cola, es decir, el exchange buscará las colas cuya routing_key coincidan con la del mensaje. Por último, el de tipo Topic también utiliza el mismo concepto del tipo direct, pero añade la posibilidad de utilizar patrones, de esta forma busca que la routing_key del mensaje coincida con el patrón establecido por la cola.

En este proyecto, se va a utilizar el exchange de tipo Topic, ya que es el que mayor escalabilidad y reutilización permite. En concreto, los patrones a utilizar son los siguientes:

- El exchange será: 'findata_symbols'.

- Para el servicio de recolección:
  1. SYMBOL_HISTORY_ROUTING_KEY = 'findata.symbol'.
  2. STOCKS_HISTORY_ROUTING_KEY = 'findata.symbol.stock'.
  3. INDEXES_HISTORY_ROUTING_KEY = 'findata.symbol.index'.
- Para el servicio de cálculos:
  1. SYMBOLS_QUEUE = 'fincalcs_symbols' como cola para conectar al exchange.
  2. SYMBOLS_TOPIC_ROUTING_KEY = 'findata.symbol.#' como patrón para captar mensajes del exchange.
  3. SYMBOLS_STOCK_ROUTING_KEY = 'findata.symbol.stock'.
  4. SYMBOLS_INDEX_ROUTING_KEY = 'findata.symbol.index'.

Por otro lado, es interesante mencionar la implementación de la conexión con MongoDB. Dado que el adaptador del repositorio de MongoDB se inicializa mucho antes de lo que se va a usar, la conexión se puede desconectar por estar inactiva. Para evitar esto, se ha utilizado la opción de conexión "perezosa" que provee Pymongo, de esta forma no comienza la conexión hasta la primera operación.

En cuanto a RabbitMQ, sucede lo mismo con la conexión. En este caso Pika no provee una opción similar a Pymongo, por lo que se ha optado por añadir una comprobación al principio de cada operación que, si la conexión no ha sido iniciada, se hace. De este modo se simula una conexión "perezosa".

Con relación a las peticiones HTTP que se realizan desde los servicios hacia terceros, un problema recurrente es que la conexión pueda fallar, ya sea a causa de la red o porque el tercero no está disponible. Para robustecer estas llamadas se ha desarrollado una clase que proporciona una implementación que mezcla las bibliotecas de *urllib3* y *requests*.

```
import requests
from requests.adapters import HTTPAdapter
from requests.exceptions import RequestException
from urllib3 import Retry

from src import settings as st


class HttpRequestException(Exception):
    pass


class HttpRequest:
    def __init__(self, status_forcelist: list[int]):
        self.session = requests.session()
        # backoff_factor time expected = [5, 10, 20, 40, 80] segs.
        retries_conf = Retry(total=5, connect=1, read=3, redirect=2, status=5, other=1,
```

```
                    backoff_factor=5, status_forcelist=status_forcelist)
        adapter = HTTPAdapter(max_retries=retries_conf)
        self.session.mount(prefix='https://', adapter=adapter)
        self.session.mount(prefix='http://', adapter=adapter)

    def get(self, url: str, timeout=None) -> requests.Response:
        try:
            if timeout is not None:
                return self.session.get(url=url, timeout=timeout)
            return self.session.get(url=url)
        except RequestException as e:
            st.logger.exception(e)
            raise
```

En el código se puede ver la implementación de esta clase. Se utiliza el objeto *Retry* de urllib3, que se configura para que en caso de que la respuesta a la petición reciba uno de los códigos HTTP establecidos en el parámetro *status_forcelist*, se inicie una serie de reintentos.

En total serán 5 reintentos para llamadas generales, si la petición es de conexión solo 1, si es de tipo lectura 3 y si es de redirección 2. En cuánto al *backoff_factor*, es una variable que se introduce en la ecuación propia de urllib3, que calcula el tiempo entre reintentos. Como se ve en el comentario cada reintento tardará de forma incremental 5, 10, 20, 40 y 80 segundos. De esta forma, si se trata de un error fortuito, se da tiempo al otro servicio para recuperarse.

Después, se configura el objeto para que de respuesta, utilizando este patrón de reintento, a todas las peticiones HTTP y HTTPS de tipo GET, que es la única operación necesaria en este caso. Aunque puede extenderse a todas las operaciones.

Para acabar este apartado, queda comentar el uso de la biblioteca estándar de logging de Python. La biblioteca establece diferentes niveles de importancia para los mensajes que se producen durante la ejecución del servicio, de menor importancia a mayor son: Notset, Debug, Info, Warning, Error y Critical.

En los tres microservicios se ha elegido implementar dos logs. El primero es *<service-name>.log*, que guarda todos los logs desde el nivel Info. El segundo log es *<service-name>_errors.log* que obtendrá los registros desde el nivel Error. Esta división permite un filtrado previo que ayuda a identificar más rápido los registros abiertos. El formato elegido para estos registros es '%(asctime)s - %(levelname)s - %(message)s', que permite ver la hora exacta, el nivel de alerta y el mensaje concreto.

Cada log puede guardarse de diferentes formas, en este caso se ha escogido guardarlos en ficheros de texto. Para evitar que estos ficheros se vuelvan demasiado grandes con el tiempo, se ha elegido usar la opción de rotación de fichero. De esta forma, cuando los ficheros alcancen 2MB de tamaño, se creará un nuevo fichero mientras que el actual se guarda. Se ha seleccionado que se puedan guardar hasta 2 ficheros antiguos. Esta opción funciona como un Buffer circular, cuando llegue un tercer guardado, el más antiguo se

borra.

7.2. Servicio de recolección de datos

Como se ha explicado en la sección 5.2 del capítulo de arquitectura, este servicio sigue un diseño hexagonal. Por ello se va a dividir esta sección en diferentes subsecciones, que irán siguiendo el flujo de ejecución de cada uno de los casos de uso que se explicarán a continuación.

Este servicio recolecta los datos a través de 4 casos de uso. Cada uno de casos de uso está configurado para que se ejecute de forma periódica.

El primero es *ObtainTickersUseCase*, este caso de uso busca los tickers de los símbolos que componen cada mercado y después los persiste en la colección de MongoDB, como se explicó en la sección 6.1. Este caso de uso se ejecuta cada sábado a las 03:30 a.m.

El segundo es *FetchSymbolsInfo*. Este caso busca la información de cada símbolo, como puede ser su ISIN, nombre, etc., de todos los símbolos que componen los mercados de acciones previamente cargados. Después persiste esta información en la colección de MongoDB correspondiente. Se ejecuta cada domingo a las 03:30 a.m.

A continuación, el lunes a las 03:30 a.m se ejecuta el caso de uso *FetchIndexesData*. Éste se usa para obtener los datos financieros de los índices bursátiles. Una vez obtenidos estos datos, se accede a la base de datos y se agrega la información de los símbolos previamente buscados. Por último se envían como mensaje al resto del sistema a través de RabbitMQ, como se muestra en la sección 6.1.

Finalmente, el lunes a las 04:30 a.m se ejecuta el caso de uso *FetchSymbolsData*. Este caso de uso es un proceso equivalente al FetchIndexesData, pero de él se obtiene los datos financieros de las acciones de cada mercado.

A continuación, en el código, primero se muestra como ejemplo de caso de uso la implementación de FetchIndexesData. El método *execute_with_scheduler* recibe por parámetro el objeto scheduler que proporciona APScheduler. Este objeto scheduler es el mismo para todos los casos de uso. También se ve como se inician todos los casos de usos en el main del servicio.

```
class FetchIndexesData(UseCaseInterface):
    def execute(self):
        """
        This use case fetches for information and financial data of the market indexes.
        """
        st.logger.info("Executing fetch symbols use case.")
        symbols_service = YFinanceAdapter(symbols_repository=MongoSymbolsAdapter(),
                                          exchanges_repository=MongoExchangesAdapter(),
                                          publisher=RabbitmqPublisherAdapter())
        indexes = symbols_service.fetch_indexes()
```

```
        symbols_service.publish_indexes(indexes)
        st.logger.info("Fetch symbols use case finished.")

    @staticmethod
    def execute_with_scheduler(scheduler: BackgroundScheduler):
        scheduler.add_job(YFinanceAdapter(symbols_repository=MongoSymbolsAdapter(),
                                          exchanges_repository=MongoExchangesAdapter(),
                                          publisher=RabbitmqPublisherAdapter())
                          .fetch_indexes,
                          'cron', day_of_week='mon',
                          hour=3, minute=30,
                          misfire_grace_time=None)


import time

from apscheduler.schedulers.background import BackgroundScheduler

from src.Exchange.application.use_cases import ObtainExchangeTickersUseCase
from src.Symbol.application.use_cases import FetchSymbolsData, FetchIndexesData,
                                             FetchSymbolsInfo
from src import settings as st

if __name__ == '__main__':
    scheduler = BackgroundScheduler(logger=st.logger)

    ObtainExchangeTickersUseCase().execute_with_scheduler(scheduler)
    FetchSymbolsInfo().execute_with_scheduler(scheduler)
    FetchIndexesData().execute_with_scheduler(scheduler)
    FetchSymbolsData().execute_with_scheduler(scheduler)

    scheduler.start()
    while True:
        time.sleep(60)
        pass
```

7.2.1. Implementación de las entidades

Antes de hablar de los casos de uso en detalle, es conveniente comentar la implementación de las entidades.

En primer lugar como se adelantó en el capítulo de modelo de datos 6.1, el servicio tiene dos entidades: Exchange y Symbol.

La entidad de Symbol representa un símbolo financiero, existen muchos tipos de símbolos, desde acciones hasta commodities como el oro, o valores de renta fija como bonos de deuda estatal. El servicio solo maneja dos tipos: las acciones y los índices bursátiles. Sin embargo, para poder mantener el servicio ampliable se ha establecido una relación

de herencia en la implementación de estas entidades.

De esta forma tenemos el objeto Symbol, que contiene como información: ticker, nombre e histórico. Este objeto representa cualquier símbolo en general. Las acciones tienen además de lo anterior, un isin y un mercado al que pertenecen. Por ello, se ha extendido Symbol con el objeto Stock que añade esta información.

Además de los anteriores, para el caso de uso FetchSymbolsInformation se ha implementado los objetos StockInformation e IndexInformation, para usarlos a modo de *Transfer* [53].

7.2.2. ObtainTickersUseCase

Este caso de uso utiliza el adaptador *Scraper* para llevar a cabo la obtención de los datos. El adaptador implementa el puerto correspondiente. Este puerto establece como interfaz dos métodos: *fetch_stock* y *create_exchange_entity*.

Por defecto, Python no posee la figura de interfaz como ocurre en lenguajes como Java. Por ello, para poder obtener el concepto de interfaz que obliga a las clases que la usan a implementar los métodos de ésta, se ha utilizado la biblioteca estándar de Python, *abc*.

En el código, se puede ver como se utiliza esta biblioteca para ”simular” el comportamiento de la interfaz que se busca.

```
from abc import ABCMeta, abstractmethod

from src.Exchange.domain.exchange import Exchange
from src.Exchange.domain.ports.exchange_repository_interface import ExchangeRepositoryInterface


class ExchangeServiceInterface(metaclass=ABCMeta):
    @classmethod
    def __subclasshook__(cls, subclass):
        return (hasattr(subclass, 'fetch_stocks') and
                callable(subclass.fetch_stocks) and
                hasattr(subclass, 'create_exchange_entity') and
                callable(subclass.create_exchange_entity)) or NotImplemented

    def __init__(self, repository: ExchangeRepositoryInterface):
        self.repository = repository

    @abstractmethod
    def fetch_stocks(self, exchanges: tuple[str, ...]):
        """
        Obtains all tickers that compounds a stock exchange.
        :param exchanges: Stock exchanges from which we want to get the tickers.
        :return: Tickers that compounds the specified stock exchange.
        """
```

```
        raise NotImplemented

    @abstractmethod
    def create_exchange_entity(self, ticker: str, symbols: tuple[str, ...]) -> Exchange:
        raise NotImplemented
```

Retomando el adaptador, cabe destacar el uso de la biblioteca *BeautifulSoap*. En el código a continuación, se muestra cómo se implementa el *scraping* de los datos a partir del código HTML correspondiente. En este caso, la web de Yahoo Finance utiliza el framework de desarrollo Front, React. Por ello, una vez localizado el elemento que se busca en el HTML, se pasa a formato JSON y a través de él se accede a la lista de componentes.

```
@staticmethod
    def __fetch_symbols(ticker: str) -> tuple[str, ...]:
        try:
            req = HttpRequest(status_forcelist=[300, 301, 400, 401, 403, 404, 408, 500, 502, 503])\
                .get(url=f'https://es.finance.yahoo.com/quote/{ticker}/components', timeout=15)
        except HttpRequestException as e:
            raise DomainServiceException()

        soup = BeautifulSoup(req.content, features='lxml')
        script = soup.find("script", text=re.compile("root.App.main"))
        # It's necessary to use json, because the page uses react for loading the data.
        data = ujson.loads(re.search("root.App.main\s+=\s+(\{.*\})", str(script)).group(1))
        tickers_data = data['context']['dispatcher']['stores']
                       ['QuoteSummaryStore']['components']['components']
        tickers = tuple(tickers_data) if tickers_data is not None else tuple()
        return tickers
```

Finalmente con la tupla de componentes obtenida, solo queda formar la entidad de tipo Exchange y pasarla al repositorio de MongoDB para que la persista.

7.2.3. FetchSymbolsInfo

En este caso de uso se busca la información que no incluye historial financiero de todos los símbolos, independientemente de su tipo. Esta información podría hacerse en un solo paso junto a los siguientes casos de uso, sin embargo, se ha decidido separarlo en un caso de uso independiente. De esta forma se gana en rendimiento de memoria al captarse menos datos, y también se ahorra tiempo en cada ejecución.

Este caso de uso utiliza la biblioteca *YFinance* [54] que ofrece operaciones para obtener mucha información de los símbolos, incluido sus datos históricos. Esta biblioteca tenía algunos errores a la hora de trabajar con símbolos de mercados no estadounidenses, esto se debe al formato de tickers que utiliza Yahoo Finance para identificar símbolos.

Yahoo Finance añade un par de letras al final del ticker para identificar el mercado en caso de que no sea estadounidense. Por ejemplo, Amazon es AMZN, el banco Santander

en España es SAN.MC y Airbus en la bolsa de París es AIR.PA. Por ello en la figura 7.1 se muestra el cambio realizado en un fork de YFinance.

```
@@ -51,9 +51,7 @@ def __init__(self, tickers):
            for ticker in self.symbols:
                ticker_objects[ticker] = Ticker(ticker)

-           self.tickers = _namedtuple(
-               "Tickers", ticker_objects.keys(), rename=True
-           )(*ticker_objects.values())
+           self.tickers = ticker_objects

        def history(self, period="1mo", interval="1d",
                    start=None, end=None, prepost=False,
@@ -88,7 +86,7 @@ def download(self, period="1mo", interval="1d",
                                     **kwargs)

            for symbol in self.symbols:
-               getattr(self.tickers, symbol)._history = data[symbol]
+               self.tickers[symbol]._history = data[symbol]

            if group_by == 'column':
                data.columns = data.columns.swaplevel(0, 1)
```

Figura 7.1: Solución al problema del ticker en el fork de YFinance.

Otro cambio necesario es el relacionado con la obtención del nombre completo de la acción. YFinance obtiene esa información de la página Business Insider [55]. En este caso la biblioteca tenía problemas gestionando los nombres que incluyen el tipo de empresa al final, por ejemplo, ”Acciona, s.a”. En la figura 7.2 se ve el cambio realizado para solucionarlo.

```
@@ -569,9 +569,9 @@ def get_isin(self, proxy=None):

            search_str = '"{}|'.format(ticker)
            if search_str not in data:
-               if q.lower() in data.lower():
+               if q.lower() in data.lower() or q.replace(",", "").lower() in data.lower():
                    search_str = '"|'
-                   if search_str not in data:
+                   if search_str not in data and '|' not in data:
                        self._isin = '-'
                        return self._isin
                else:
```

Figura 7.2: Solución al problema del nombre en el fork de YFinance.

Por último, cabe comentar que la decisión de utilizar un fork y no el proyecto de YFinance original es que permite tener independencia e inmediatez a la hora de realizar

cambios.

Una vez realizados estos arreglos, en el adaptador del servicio se introduce una regla de negocio para descartar todos los símbolos que aun así sigan sin tener un nombre o isin válidos.

A continuación se puede ver el código relacionado con la regla anterior y, además, el uso de segmentos (chunks) en los símbolos para evitar sobrecargar las llamadas que realiza la librería. De otra forma, tanto Yahoo Finance como Business Insider dejarían de dar servicio a las peticiones por haberlos sobrecargado.

```
def __fetch_symbols_info(self, symbol_tickers: tuple[str, ...], exchange_ticker: str) -> None:
    """
    Fetch all the symbols info as name and isin using yfinance, and then stores it into db.
    """
    chunks = self.__get_chunks(symbol_tickers)
    for chunk in chunks:
        symbols = []
        st.logger.info("Fetching {} symbols info with tickers: {}".format(len(chunk), chunk))
        for ticker in chunk:
            try:
                symbol_data = yf_ticker_info(ticker)
                name = symbol_data.info.get('shortName')
                if name is None:
                    name = symbol_data.info.get('longName', '-')
                if ticker in st.exchanges:
                    if name == '-':
                        st.logger.info("Discarding symbol: {}".format(ticker))
                    else:
                        name = self.__format_name_field(name)
                        symbols.append(IndexInformation(ticker=ticker, name=name))
                else:
                    isin = symbol_data.isin
                    if name == '-' or isin == '-':
                        st.logger.info("Discarding symbol: {}".format(ticker))
                    else:
                        name = self.__format_name_field(name)
                        symbols.append(StockInformation(ticker=ticker, name=name, isin=isin,
                                                        exchange=exchange_ticker))
            except Exception as e:
                st.logger.exception(e)
            time.sleep(1)
        try:
            self.symbols_repository.save_symbols_info(symbols=tuple(symbols))
        except RepositoryException:
            raise DomainServiceException()
```

La forma de dividir los símbolos en segmentos, de 1000 como máximo, se muestra a continuación.

```
@staticmethod
    def ___get_chunks(symbols_tickers: tuple[str, ...]) -> tuple[tuple[str, ...]]:
        # Divides symbols_tickers in chunks for avoid to overload yahoo finance api
        chunk_len = 1000 if len(symbols_tickers) > 1000 else len(symbols_tickers)
        chunks = tuple(symbols_tickers[i:i + chunk_len] for i in
                       range(0, len(symbols_tickers), chunk_len))
        return chunks
```

Finalmente, una vez obtenida la información de todos los símbolos, se persiste en MongoDB mediante el repositorio correspondiente.

7.2.4. FetchIndexesData y FetchSymbolsData

Dado que las diferencias entre estos dos casos de uso son mínimas y reutilizan casi todo el código, se van a explicar juntos.

En estos casos de uso, primero se obtiene la información de cada símbolo persistida anteriormente en la base de datos. Después se obtiene su histórico financiero de YFinance y finalmente, se crean las instancias de tipo Symbol correspondientes, en este caso pueden ser Stock o Index dependiendo del caso de uso.

Para concluir, se publica esta información a través de RabbitMQ al final de cada uno de los casos de uso. A continuación, se adjunta respectivamente el código de la publicación de símbolos, Stocks o de otro tipo y de símbolos de tipo Índice.

```
def publish_symbols(self, symbols: tuple[Symbol, ...]) -> None:
        if self.connection is None:
            self.connection = self.___connect_to_rabbit()

        st.logger.info("Publishing symbols: {}".format([s.ticker for s in symbols]))
        conn_channel = self.connection.channel()
        for symbol in symbols:
            adapted_historic = self.___adapt_stock_historic(symbol.historical_data)
            if isinstance(symbol, Stock):
                message = {
                    'ticker': symbol.ticker,
                    'isin': symbol.isin,
                    'name': symbol.name,
                    'historic': adapted_historic,
                    'exchange': symbol.exchange
                }
                routing_key = st.STOCKS_HISTORY_ROUTING_KEY

            else:
                message = {
                    'ticker': symbol.ticker,
                    'name': symbol.name,
                    'historic': adapted_historic
                }
```

```
            routing_key = st.SYMBOL_HISTORY_ROUTING_KEY

        message = ujson.dumps(message)
        try:
            conn_channel.basic_publish(exchange=st.SYMBOLS_HISTORY_EXCHANGE,
                                       routing_key=routing_key,
                                       body=message)
        except Exception as e:
            st.logger.exception(e)
            pass

def publish_indexes(self, indexes: tuple[Symbol, ...]) -> None:
    if self.connection is None:
        self.connection = self.__connect_to_rabbit()

    st.logger.info("Publishing indexes: {}".format([s.ticker for s in indexes]))
    conn_channel = self.connection.channel()
    for index in indexes:
        adapted_historic = self.__adapt_index_historic(index.historical_data)
        message = {
            'ticker': index.ticker,
            'name': index.name,
            'historic': adapted_historic
        }
        message = ujson.dumps(message)
        try:
            conn_channel.basic_publish(exchange=st.SYMBOLS_HISTORY_EXCHANGE,
                                       routing_key=st.INDEXES_HISTORY_ROUTING_KEY,
                                       body=message)
        except Exception as e:
            st.logger.exception(e)
            pass
```

7.3. Servicio de cálculos financieros

Como se explicó previamente en el capítulo de arquitectura 5.3, este servicio utiliza un diseño hexagonal para permitir desacoplar sus operaciones del dominio, de las tecnologías concretas. El servicio cuenta con dos partes, la primera es el consumidor de RabbitMQ que permite la ingesta de datos financieros. La segunda parte es una API Rest que permite exponer estos datos a cualquier cliente del sistema. Por ello, esta sección se va a dividir en una primera subsección, 7.3.1, para explicar las métricas y cálculos que realiza el servicio. La subsección 7.3.2 para exponer la implementación del dominio. La 7.3.3 para la parte del consumidor de RabbitMQ, y por último, la subsección 7.3.4 para la API.

7.3.1. Métricas y cálculos financieros

El servicio tiene como objetivo principal calcular métricas de rendimiento y riesgo financiero sobre los símbolos financieros que posee, y sobre los escenarios de cartera de inversión que los clientes le soliciten.

Las métricas que se ha decidido calcular son las siguientes:

- **Compound Annual Growth Rate**:
  Es el ratio que mide el crecimiento de la rentabilidad, para un período de tiempo determinado asumiendo que se reinvierten las ganancias al final de cada período. En este servicio se ofrece el cálculo para 3 y 5 años.
  Su fórmula es:
  $$\left(\frac{V_f}{V_i}\right)^{1/t} - 1 \tag{7.1}$$
  Donde Vf es el valor final del periodo, Vi el inicial y t es el periodo.

- **Rentabilidad**: Es la variación diaria entre el precio actual y el precio anterior, es decir, la ganancia o pérdida diaria de un activo. Su fórmula es:
  $$\text{Return } = \frac{\text{Current market value-Original investment value}}{\text{Original investment value}} \tag{7.2}$$
  Cabe destacar que en el caso de las carteras de inversión, el valor de un retorno diario es el sumatorio de las rentabilidades de cada acción ponderada al peso de la acción en la cartera. Donde el peso es el porcentaje que representa la acción de toda la cartera. Cuya fórmula es:
  $$\mathrm{R_p} = \sum_{i=1}^{n} \mathrm{W}_i \mathrm{R_i} \tag{7.3}$$

- **Rentabilidad Anualizada**: Se trata de los retornos de cualquier periodo escalados a un periodo de 12 meses. De esta forma se permite comparar diferentes activos sin importar sus periodos reales. En este servicio esta métrica se aplica a las carteras de inversión. Su fórmula es:
  $$\mathrm{Ra} = (1 + (\mathrm{Rf} - \mathrm{Ri})/\mathrm{Ri})\text{^}(12/\mathrm{n}) - 1 \tag{7.4}$$
  Donde Ri es la rentabilidad inicial del periodo, Rf la final, y n el número de meses total del periodo.

- **Volatilidad**: Es la medida de las fluctuaciones de los precios o rentabilidades de un activo, es decir, es una medida del riesgo pasado del activo. El servicio ofrece este cálculo para las carteras de inversión. El cálculo se basa en la desviación típica, por ello la fórmula es:
  $$\sigma = \sqrt{\frac{\sum_i^N \left(X_i - \bar{X}\right)^2}{N}} \tag{7.5}$$

- **Volatilidad anualizada**: La anualización permite tener una aproximación de la volatilidad que se puede esperar en un año de ese activo. Se ha decidido que el factor de anualización sea 252, que es el número de días de mercado abierto al año. La fórmula es:

$$Vol_A = Vol_D * \sqrt{Annualization factor} \tag{7.6}$$

  Donde VolD son las volatilidades diarias.

- **Maximum Drawdown**: El Maximum Drawdown muestra el mayor movimiento del valor desde un máximo hasta un mínimo. Es decir, proporciona el valor de la máxima pérdida histórica. Su fórmula es:

$$MDD = \frac{\text{Min Value } - \text{ Max Value}}{\text{Max Value}} \tag{7.7}$$

- **Sharpe Ratio**: Este ratio permite comparar el retorno esperado de la inversión con su riesgo asociado. Para calcularlo se necesita los retornos del portfolio, la tasa libre de riesgo y la volatilidad. La tasa libre de riesgo representa el retorno teórico de una inversión sin riesgo, como por ejemplo, un bono del tesoro. Este ratio se ha supuesto como un 2 %.
  La fórmula es:

$$\text{Sharpe Ratio } = \frac{R_p - R_f}{\sigma_p} \tag{7.8}$$

- **Sortino ratio**: Este ratio es una alternativa al Sharpe ratio. En el primero, se toma la volatilidad total, es decir, que también incluye las desviaciones positivas. Por contra, este ratio supone que el inversor solo se ve afectado por las desviaciones negativas, y por tanto, se calcula usando la desviación típica negativa. Su fórmula es:

$$\text{Sortino Ratio } = \frac{R_p - R_t}{\sigma_{neg}} \tag{7.9}$$

  Donde Rt es el retorno objetivo, es decir, los retornos de un índice de referencia elegido arbitrariamente para comparar el resultado del Portfolio.

- **Calmar ratio**: Este ratio persigue el mismo objetivo que el Sharpe ratio, sin embargo, utiliza como medida del riesgo el Maximum Drawdown y no la volatilidad. Su fórmula es:

$$\text{Calmar ratio } = \frac{R_p - R_f}{D_{\text{Max}}} \tag{7.10}$$

7.3.2. Implementación del dominio

Este servicio tiene dos entidades definidas: Symbol, para representar los símbolos financieros y Portfolio, para representar la agrupación de varios Symbols en una cartera de inversión, y proporcionar un análisis de riesgo y rendimiento.

Respecto a los Symbol, el servicio busca consumir símbolos de tipo índice bursátil y acción, aunque no se cierra la puerta a ampliar funcionalidad, y por tanto, soportar nuevos tipos de símbolo en el futuro. Para garantizar esto, se ha implementado una herencia de objetos.

El objeto base es Symbol, que contiene ticker, nombre, cierres diarios y rentabilidades diarias. Este Symbol es extendido por los objetos Index y Stock. En el caso de Index, no añade nueva información, por lo que es idéntico. En cuanto al Stock, añade información de los dividendos, isin y mercado al que pertenece (exchange). Además de estas propiedades, la clase Symbol contiene la lógica para extraer los datos de cierres y rentabilidades desde un objeto Pandas, y utiliza el decorador *property* de Python, para obtener la primera y última fecha del histórico. El decorador permite tratar a estas operaciones como si fuesen propiedades del objeto. Todo esto se muestra en el siguiente código:

```
class Symbol:
    def __init__(self, ticker: str, name: str, closures: dict, daily_returns: dict = None):
        self.ticker = ticker
        self.name = name
        self.closures, processed_daily_returns = self._process_historical_data(closures, daily_returns)
        self.daily_returns = (self._compute_daily_returns()
                              if processed_daily_returns is None else processed_daily_returns)

    @staticmethod
    def _process_historical_data(closures: dict, daily_returns: dict = None) -> tuple[pd.Series,
                                                                                       Union[pd.Series, None]]:
        """
        :param closures: closures of the symbol as dict
        :param daily_returns: (optional) daily_returns of the symbol as dict, if None,
               would be computed.
        :return: historic data of the symbol with properly pd.Series
        """
        try:
            indexes = tuple(datetime.strptime(i, '%Y-%m-%d %H:%M:%S').date()
                            for i in tuple(closures.keys()))
        except ValueError:
            indexes = tuple(datetime.strptime(i, '%Y-%m-%d').date()
                            for i in tuple(closures.keys()))

        pd_closures = pd.Series(data=closures.values(), index=indexes)
        pd_closures.index = pd.to_datetime(pd_closures.index)
        if daily_returns is not None:
            pd_daily_returns = pd.Series(data=daily_returns.values(), index=indexes)
            pd_daily_returns.index = pd.to_datetime(pd_daily_returns.index)
            return pd_closures, pd_daily_returns

        return pd_closures, None

    @property
    def first_date(self):
        return self.closures.index[0]

    @property
    def last_date(self):
        return self.closures.index[-1]
```

```
    def _compute_daily_returns(self) -> pd.Series:
        self.daily_returns = self.closures.pct_change()
        return self.daily_returns
```

Para acabar con la implementación de los Symbol, queda hablar del servicio del dominio. Esta clase se utiliza en arquitectura hexagonal para encapsular la lógica creacional de las entidades y también para operaciones que están estrechamente relacionadas con la entidad, pero cuya lógica no encaja en la responsabilidad de la propia entidad.

En el código se muestra la implementación del servicio. En la operación de creación se puede ver como este servicio de dominio desacopla la lógica creacional de las entidades de los clientes. También se puede observar el método *compute_cagr* que implementa la métrica de *Compound Annual Growth Rate* 7.1.

```
class DomainService:
    @staticmethod
    def create_symbol_entity(ticker: str, closures: dict, name: str,
                             isin: str = None, exchange: str = None,
                             dividends: dict = None, daily_returns: dict = None) -> Symbol:
        if dividends is not None and exchange is not None:
            return Stock(ticker=ticker, isin=isin, name=name, closures=closures,
                         dividends=dividends, daily_returns=daily_returns, exchange=exchange)
        elif ticker in st.EXCHANGES:
            return Index(ticker=ticker, name=name, closures=closures, daily_returns=daily_returns)

        else:
            return Symbol(ticker=ticker, name=name, closures=closures, daily_returns=daily_returns)

    @staticmethod
    def compute_cagr(entity: Union[Index, Stock], period: Literal['3yr', '5yr'] = '3yr') -> float:
        """
        Compound annual growth rate
        :param entity: Entity for which compute the cagr.
        :param period: could be '3yr' or '5yr'
        """
        if period not in ["3yr", "5yr"]:
            raise AttributeError

        today = entity.closures.index[-1]
        today = datetime(today.year, today.month, today.day)
        if period == '3yr':
            n = 3
            first_date = datetime(today.year - 3, today.month, today.day)
        elif period == '5yr':
            n = 5
            first_date = datetime(today.year - 5, today.month, today.day)

        closes = entity.closures[entity.closures.index >= first_date]
```

```
        cagr = ((closes[-1] / closes[0]) ** (1 / n)) - 1
        return cagr
```

Respecto a los cálculos realizados para el análisis del Portfolio, su implementación se encuentra distribuida entre la entidad de Portfolio y su servicio del dominio. En la entidad se realizan los cálculos base, es decir, los referentes a los retornos y a la volatilidad. A continuación, se irá comentando la implementación de la entidad por partes.

El primer código corresponde al constructor y a la forma de obtener la ventana temporal sobre la que se realizará el análisis:

```
class Portfolio:
    def __init__(self, symbols: tuple[Symbol], n_shares_per_symbol: dict[str, int],
                 initial_date: Union[datetime.date, None], end_date: Union[datetime.date,
                 ↪ None]):
        self.symbols = symbols
        self.total_shares = sum(n_shares_per_symbol.values())
        self.weights = {symbol.ticker: (n_shares_per_symbol[symbol.ticker] /
        ↪ self.total_shares)
                        for symbol in symbols}
        self.first_date, self.last_date = self.__compute_common_date(initial_date,
        ↪ end_date)

    def __compute_common_date(self, initial_date, end_date) -> tuple:
        common_idx = self.symbols[0].daily_returns.index
        for symbol in self.symbols:
            common_idx = common_idx.intersection(symbol.daily_returns.index)
        if initial_date is not None \
                and (datetime.date(day=common_idx[0].day,
                                   month=common_idx[0].month,
                                   year=common_idx[0].year) < initial_date <
                                   ↪ datetime.date(day=common_idx[-1].day,
                                                   month=common_idx[-1].month,
                                                   year=common_idx[-1].year)):
            common_idx = common_idx[common_idx
                                    .slice_indexer(initial_date.strftime("%Y-%m-%d"),
                                    ↪ common_idx[-1])]

        if end_date is not None \
                and datetime.date(day=common_idx[0].day,
                                  month=common_idx[0].month,
                                  year=common_idx[0].year) < end_date <
                                  ↪ datetime.date(day=common_idx[-1].day,
                                                  month=common_idx[-1].month,
                                                  year=common_idx[-1].year):
            common_idx = common_idx[common_idx.slice_indexer(common_idx[0],
            ↪ end_date.strftime("%Y-%m-%d"))]

        return common_idx[0], common_idx[-1]
```

El método *__compute_common_date* obtiene la primera fecha válida más cercana a la fecha inicial dada por el usuario, y la fecha válida más cercana a la fecha final dada por el usuario. Por tanto, el objetivo de este algoritmo es obtener la ventana temporal válida más aproximada a la deseada por el usuario.

A continuación, se muestra el código relacionado al cálculo de los retornos:

```
@property
def weighted_returns(self) -> pd.Series:
    weighted_rets =
    ↪  DataFrame(data=[symbol.daily_returns[self.first_date:self.last_date]
                       .rename(index=symbol.ticker, inplace=True) *
                       ↪  self.weights[symbol.ticker]
                          for symbol in self.symbols]).transpose()
    return weighted_rets.sum(axis=1)

@property
def annualized_returns(self):
    start_date = self.weighted_returns.index[0]
    end_date = self.weighted_returns.index[-1]
    months_passed = (end_date - start_date) / np.timedelta64(1, "M")

    total_return = (self.weighted_returns[-1] - self.weighted_returns[1]) /
    ↪  self.weighted_returns[1]
    total_return_arr = np.array([1+total_return])

    return (np.float_power(abs(total_return_arr),
                           np.array([12/months_passed]))*np.sign(total_return_arr)) - 1
```

Como se puede ver, se ha hecho uso del decorador *property* al igual que en Symbol. Para calcular los retornos ponderados (weighted_returns), se obtienen los retornos diarios de cada acción dentro de la ventana temporal y se les aplica su peso en la cartera. Finalmente, se devuelve el retorno acumulado de cada acción.

Para anualizar las rentabilidades, primero se calcula los meses que contiene la ventana temporal. A continuación se obtiene el retorno total del periodo y se transforma a una array de Numpy. Finalmente se aplica la ecuación 7.4 haciendo uso de Numpy.

El siguiente código corresponde a los cálculos relacionados con la volatilidad del Portfolio:

```
@property
def volatility(self):
    return self.weighted_returns.std()

@property
def annualized_volatility(self):
    return self.volatility * math.sqrt(st.ANNUALIZATION_FACTOR)
```

```
@property
def mdd(self):
    """
    Max Drawdown
    """
    cum_rets = self.weighted_returns.add(1).cumprod()
    nav = ((1 + cum_rets) * 100).fillna(100)
    hwm = nav.cummax()
    dd = nav / hwm - 1
    return min(dd)
```

Para calcular el Maximum Drawdown, es necesario obtener el volúmen del portfolio (nav) y a partir de él obtener el drawdown. Finalmente se devuelve el valor mínimo de este drawdown, que representará el mínimo absoluto y por tanto el maximum drawdown.

Por último, en el servicio del dominio se implementan los cálculos de los ratios de Sharpe, Sortino y Calmar. El código es el siguiente:

```
@staticmethod
def sharpe_ratio(entity: Portfolio):
    return float(((entity.annualized_returns - st.RISK_FREE_RATIO) /
    ↪  entity.annualized_volatility)[0])

@staticmethod
def sortino_ratio(entity: Portfolio, benchmark_returns: pd.Series):
    neg_asset_returns = (entity.weighted_returns[entity.weighted_returns <
    ↪  0][entity.first_date:entity.last_date])
    std_dev = neg_asset_returns.std()
    return float(((entity.annualized_returns - benchmark_returns.mean())
                  / (std_dev * np.sqrt(st.ANNUALIZATION_FACTOR)))[0])

@staticmethod
def calmar_ratio(entity: Portfolio):
    return float(((entity.annualized_returns - st.RISK_FREE_RATIO) /
    ↪  entity.mdd)[0])
```

7.3.3. Implementación del consumidor de RabbitMQ

La forma que este servicio utiliza para captar la información que el servicio de recolección provee es a través de RabbitMQ, como se ha explicado previamente en la sección 7.1. Para ello, como se puede observar en la sección 5.3 del capítulo de arquitectura, el servicio cuenta con dos puertos, un adaptador y un consumidor de RabbitMQ.

Para empezar, es necesario explicar la implementación del consumidor de RabbitMQ, que será el encargado de dar inicio a este caso de uso. Para su implementación se ha definido un puerto, *data_consumer*, que define en su interfaz las operaciones de *connect* y *disconnect* para gestionar el estado de la conexión a la cola de RabbitMQ. También establece la operación de *on_message*, que será la encargada de recibir el mensaje que

llegue de la cola y pasarlo al servicio correspondiente. La implementación de este puerto la realiza *rabbitmq_consumer*.

De esta clase, cabe destacar que se ha configurado el constructor para que los datos referentes al exchange, la cola y la routing_key sean parametrizables, por lo que el consumidor es reutilizable si se quisieran ampliar los casos de uso del servicio.

En el siguiente fragmento se muestra el código de las operaciones de *connect* y *disconnect*:

```
def connect(self) -> BlockingConnection:
    """
    Opens the connection to rabbit.
    """

    credentials = PlainCredentials(username=st.RABBIT_USER, password=st.RABBIT_PASSW)
    try:
        connection = BlockingConnection(
            ConnectionParameters(host=st.RABBIT_HOST, port=st.RABBIT_PORT,
                                 virtual_host=st.RABBIT_VHOST, credentials=credentials,
                                 connection_attempts=5,
                                 retry_delay=3))
    except (AMQPConnectionError, socket.gaierror) as e:
        st.logger.exception(e)
        self.connected = False
        raise DataConsumerException()

    st.logger.info("Symbol rabbitmq consumer connected")
    self.connected = True

    return connection

def disconnect(self):
    try:
        self.connection.close()
        st.logger.info("Symbol rabbitmq consumer disconnected")
    except ConnectionWrongStateError:
        pass
    finally:
        self.connected = False
```

Como se puede ver, se utiliza una conexión bloqueante que es la menos compleja, ya que no necesita de bibliotecas asíncronas adicionales. Esto es importante porque implica que si se quiere ejecutar el consumidor de forma asíncrona será necesario utilizar un hilo específico para el consumidor. Por ello, se implementa en un método específico de este adaptador, *start_consumer*:

```
def start_consumer(self):
    self.connection = self.connect()
```

```
    try:
        self.channel = self.__setup_consumer()
    except DataConsumerException as e:
        self.disconnect()
        raise e

    self.channel.basic_consume(on_message_callback=self.__on_message,
                               queue=self.__rabbit_queue)

    thread = threading.Thread(target=self.channel.start_consuming)
    thread.start()

def __setup_consumer(self) -> BlockingChannel:
    retry = 0
    while retry < 3:
        try:
            channel = self.connection.channel()

            channel.queue_declare(queue=self.__rabbit_queue)
            channel.basic_qos(prefetch_count=1)
            channel.queue_bind(exchange=self.__rabbit_exchange, queue=self.__rabbit_queue,
                               routing_key=self.__rabbit_routing_key)
        except AMQPChannelError as e:
            st.logger.exception(e)
            self.connected = False
            retry += 1
        else:
            self.connected = True
            return channel
    else:
        raise DataConsumerException()
```

Respecto a las implementaciones del puerto *driven_service* y de su adaptador *RabbitMQ_service*. El puerto define como interfaz las operaciones de *fetch_symbol_data*, *save_stock* y *save_index*. Por ello, como se ve en el diagrama de la figura 5.3 presentado en el capítulo de arquitectura, va a tener dependencias con el repositorio, implementado por el adaptador de MongoDB y con el *data_consumer* explicado previamente.

Como se ha expuesto previamente, se va a asociar un *data_consumer* para recibir los mensajes de RabbitMQ. Dado que éste va a iniciarse en un hilo propio, para poder comunicar los mensajes es necesario utilizar una cola. En su biblioteca estándar, Python ofrece la estructura de datos Queue, que es la que se va a usar aquí. Como se puede ver en el código, la cola ofrece un método GET que bloquea la lectura cada 0.5 segundos, y en caso de que no este vacía, se procesa el mensaje.

```
def __create_rabbit_consumer(self, rabbit_queue: str,
                             exchange: str, routing_key: str) -> RabbitmqConsumer:
    return RabbitmqConsumer(messages_received_queue=self.__consumers_queue,
                            rabbit_queue=rabbit_queue, exchange=exchange, routing_key=routing_key)
```

```
def fetch_symbol_data(self) -> None:
    """
    Gets the symbol data, converts it to a symbol entity,
    precalculates it's financials data, and saves into the db.
    """
    try:
        self.consumer.start_consumer()
    except DataConsumerException:
        raise ServiceException()

    while True:
        try:
            symbol_message = self.__consumers_queue.get(timeout=0.5)
        except queue.Empty:
            if not self.consumer.connected:
                break
            continue
        else:
            try:
                self.__process_symbol_data_message(symbol_message)
            except MessageNotValid:
                continue
```

En el siguiente código se muestra también el método de procesado del mensaje. Primero se válida el mensaje, si no contiene las claves esperadas no se procesa. En caso contrario, se decide en función de su routing_key si es de tipo Stock o de tipo Index y finalmente, se persiste usando el repositorio de MongoDB.

```
def __process_symbol_data_message(self, symbol_message: dict) -> None:
        validation = self.__validate_message_format(symbol_message)

        if not validation[0]:
            st.logger.error("Message from {} received with missing key: {}"
            .format(st.SYMBOLS_QUEUE, validation[1]))
            raise MessageNotValid()

        if symbol_message['routing_key'] == st.SYMBOLS_STOCK_ROUTING_KEY:
            self.save_stock(symbol_message)
        elif symbol_message['routing_key'] == st.SYMBOLS_INDEX_ROUTING_KEY:
            self.save_index(symbol_message)
```

7.3.4. Implementación de la API

Como también se explicó en el capítulo de arquitectura 5.3, la API de este servicio se ha implementado usando Flask. En esta subsección se va a exponer como se ha implementado tanto la parte de Flask, como los puertos y adaptadores para poder utilizarse desde la API.

En primer lugar, para que la API pueda funcionar hay que instanciar el objeto Flask, que es el objeto principal que ofrece este framework y que inicializa todos sus elementos internos. Para ello, en el paquete del proyecto llamado api, se implementa el siguiente código en el fichero *__init__.py*.

```
from flask import Flask

from src.api.portfolio_routes import portfolio_blueprint
from src.api.symbol_routes import symbols as symbols_blueprint
from src import settings as st

app = Flask(__name__)
app.register_blueprint(symbols_blueprint)
app.register_blueprint(portfolio_blueprint)


def start_api():
    app.run(host=st.API_HOST, port=st.API_PORT)
```

Flask permite estructurar las API en elementos modulares llamados *blueprints*. Estas plantillas contienen el mapeado de URL específico a cada recurso. En este servicio se ha creado un blueprint para cada entidad.

Para Symbol se han habilitado operaciones de lectura (GET) que permiten obtener todas las acciones, índices bursátiles o un símbolo en concreto mediante su ticker. Con el decorador route se establece la URL y el tipo de método HTTP.

```
symbols = Blueprint(name='symbols', import_name=__name__, url_prefix='/symbols')

symbol_service = FlaskServiceAdapter(repository=MongoRepositoryAdapter(),
                                     domain_service=DomainService())

@symbols.route('/stocks', methods=['GET'])
def get_stocks_list():
    symbs = ujson.dumps([symbol.to_json() for symbol in symbol_service.get_stocks_info()])
    return Response(response=symbs, status=200, mimetype='application/json')


@symbols.route('/indexes', methods=['GET'])
def get_indexes_list():
    indexes = ujson.dumps([index.to_json() for index in symbol_service.get_indexes_info()])
    return Response(response=indexes, status=200, mimetype='application/json')


@symbols.route('/<symbol_ticker>', methods=['GET'])
def get_symbol(symbol_ticker):
    symbol = symbol_service.get_symbol(symbol_ticker)
    if not symbol:
        return Response(response='Error: symbol not found', status=404,
```

```
                    mimetype='application/json')

    symbol = ujson.dumps(symbol.to_json())
    return Response(response=symbol, status=200, mimetype='application/json')
```

Para la parte de Portfolio, se ha implementado un único método de tipo POST. Este método es el encargado de realizar el análisis de un Portfolio para un escenario dado por el usuario. Aquí, es interesante destacar el uso de la biblioteca Cerberus para poder validar correctamente la entrada dada por el usuario. En el schema, se especifican las reglas que deben cumplir los parámetros que llegan en el body de la petición. Los tickers deben ser una lista de strings, con al menos un elemento y no puede ser nulo ni vacío. Las fechas son transformadas desde el string de entrada a un objeto date de Python. Más adelante, se implementan otras reglas de forma manual para complementar la validación.

```
portfolio_blueprint = Blueprint(name='portfolio', import_name=__name__,
                                url_prefix='/portfolio')

portfolio_service = FlaskServiceAdapter(symbol_repository=MongoRepositoryAdapter(),
                                        symbol_domain_service=SymbolDomainService(),
                                        domain_service=DomainService())


@portfolio_blueprint.route('', methods=['POST'])
def get_portfolio_analysis():
    def to_date(d):
        return datetime.strptime(d, '%d-%m-%Y').date()

    schema = {
        'tickers': {
            'type': 'list',
            'schema': {'type': 'string', 'min': 1},
            'nullable': False,
            'empty': False
        },
        'initial_date': {'type': 'date', 'coerce': to_date, 'required': False},
        'end_date': {'type': 'date', 'coerce': to_date, 'required': False}
    }
    data = request.data if len(request.data) > 0 else request.form
    try:
        data = ujson.loads(data)
    except TypeError as e:
        pass

    body = {'tickers': [ticker.strip() for ticker in data.get('tickers').split(",")]}
    initial_date = data.get('initial_date')
    end_date = data.get('end_date')
    if initial_date is not None:
        body['initial_date'] = initial_date
    if end_date is not None:
```

```
        body['end_date'] = end_date

    shares_per_stock = {}
    sps_input = data.get('sharesPerStock').split(",")
    for stock in sps_input:
        s = stock.split(":")
        shares_per_stock[s[0].strip()] = int(s[1])
    body['shares_per_stock'] = shares_per_stock
    try:
        v = Validator(schema=schema)
        v.allow_unknown = True
        val = v.validate(body, schema)

        if not val:
            return Response(response="Invalid request: {}"
                            .format(ujson.dumps(v.errors)), status=400, mimetype='application/json')

        body = v.normalized(body)
        tickers = body['tickers']
        shares_per_stock = list(body['shares_per_stock'].keys())

        if len(tickers) != len(shares_per_stock):
            return Response(response="Invalid request: number of tickers not equals "
                                     "to number of shares_per_stock's keys indicated", status=400,
                            mimetype='application/json')
        if tickers != shares_per_stock:
            return Response(response="Invalid request: Tickers and shares_per_stock's
                            keys should match.", status=400, mimetype='application/json')

        if body['initial_date'] >= body['end_date']:
            return Response(response="Invalid request: Initial Date
                            must be previous to End Date.", status=400, mimetype='application/json')
    except ValidationError as e:
        return Response(response='Invalid request: tickers not valid', status=400,
                        mimetype='application/json')

    try:
        portfolio_info = (portfolio_service.create_portfolio(tickers=tuple(body['tickers']),
                                                             n_shares_per_symbol=body['shares_per_stock'],
                                                             initial_date=body['initial_date'],
                                                             end_date=body['end_date']))
    except PortfolioException as e:
        if e.error == 'No symbols found':
            return Response(response=ujson.dumps(e.error), status=404,
                            mimetype='application/json')
        elif e.error == 'Invalid ticker':
            return Response(response=ujson.dumps(e.error), status=400,
                            mimetype='application/json')
```

```
    else:
        return Response(response=ujson.dumps(portfolio_info.to_json()), status=200,
                        mimetype='application/json')
```

En cuanto a los adaptadores, para ambas entidades funcionan como Servicios de aplicación adaptando la información de entrada que llega desde la vista y ejecutando los correspondientes métodos de los Servicios del dominio.

Para finalizar esta parte, hay que mencionar el uso de diferentes objetos siguiendo el patrón Transfer [53] para permitir la traducción de la información a objetos JSON, que serán los enviados en las respuestas. Por ejemplo:

```
@dataclass
class SymbolTransfer:
    """
    first_date: Year%month%day%
    last_date: Year%month%day%
    closures: {Year%month%day%: float}
    daily_returns: {Year%month%day%: float}
    """
    ticker: str
    name: str
    first_date: datetime.timestamp
    last_date: datetime.timestamp
    closures: dict
    daily_returns: dict

    def to_json(self):
        json = {'ticker': self.ticker, 'name': self.name,
                'first_date': self.first_date.strftime('%d-%m-%Y'),
                'last_date': self.last_date.strftime('%d-%m-%Y'),
                'closures': {k.strftime('%d-%m-%Y'): str(round(v, 4)).replace('nan', 'null')
                             for k, v in self.closures.items()},
                'daily_returns': {k.strftime('%d-%m-%Y'): str(round(v, 4)).replace('nan', 'null')
                                  for k, v in self.daily_returns.items()}}
        return json


@dataclass
class SymbolStatisticsTransfer(SymbolTransfer):
    """
    cagr: {"3yr": float, "5yr": float}
    """
    cagr: dict

    def to_json(self):
        json = super(SymbolStatisticsTransfer, self).to_json()
        json['cagr'] = {k: str(round(v, 4)) for k, v in self.cagr.items()}
        return json
```

Por último, para terminar la subsección, queda hablar de la documentación que se ha desarrollado para la propia API. En el capítulo de tecnologías usadas 4.3.3, se ha expuesto el uso de OpenAPI en su versión 3.0 para facilitar al usuario una documentación completa. Se puede ver en las figuras 7.3, 7.4 y 7.5 una parte del resultado del fichero YAML.

Figura 7.3: Resultado de la especificación de OpenAPI.

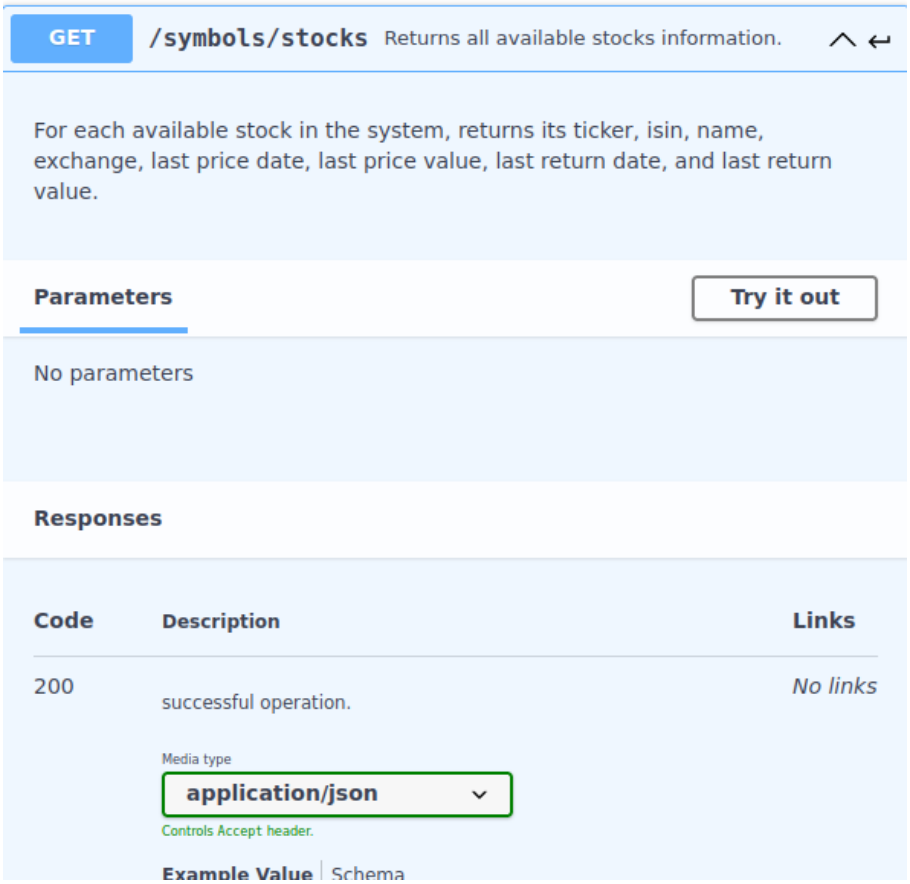


Figura 7.4: Resultado de la especificación de OpenAPI.

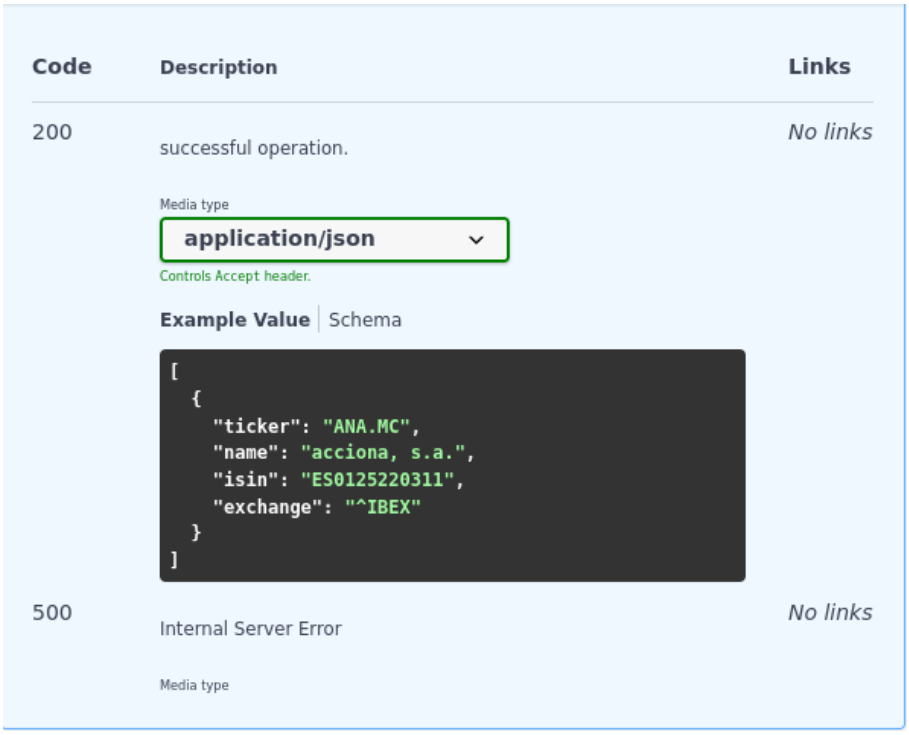


Figura 7.5: Resultado de la especificación de OpenAPI.

7.4. Aplicación Web

Como se explicaba en el capítulo de arquitectura 5.4, la aplicación se ha implementado con Django y por tanto sigue el patrón MVT. En este capítulo se va a comentar con detalle la configuración seguida y la implementación de las partes que componen la web, la aplicación de *profiles* y la de *markets*.

7.4.1. Configuración de Django

Todo proyecto de Django tiene una serie de elementos que se deben configurar. Para ello se utiliza el fichero *settings.py* que se genera de forma automática en la carpeta del proyecto. En el siguiente fragmento de código se muestran las opciones de Django definidas para la aplicación.

```
# SECURITY WARNING: keep the secret key used in production secret!
SECRET_KEY = env('SECRET_KEY')

# SECURITY WARNING: don't run with debug turned on in production!
DEBUG = env("DEBUG")

ALLOWED_HOSTS = env("ALLOWED_HOSTS")


# Application definition

INSTALLED_APPS = [
    'profiles.apps.ProfilesConfig',
    'markets.apps.MarketsConfig',
    'django.contrib.admin',
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.messages',
    'django.contrib.staticfiles',
    'chartjs',
]

MIDDLEWARE = [
    'django.middleware.security.SecurityMiddleware',
    'django.contrib.sessions.middleware.SessionMiddleware',
    'django.middleware.common.CommonMiddleware',
    'django.middleware.csrf.CsrfViewMiddleware',
    'django.contrib.auth.middleware.AuthenticationMiddleware',
    'django.contrib.messages.middleware.MessageMiddleware',
    'django.middleware.clickjacking.XFrameOptionsMiddleware',
]

ROOT_URLCONF = 'investapp.urls'
```

```
TEMPLATES = [
    {
        'BACKEND': 'django.template.backends.django.DjangoTemplates',
        'DIRS': [(os.path.join(BASE_DIR, 'templates')),
                 (os.path.join(os.path.join(BASE_DIR, 'profiles'), 'templates'))
                 ],
        'APP_DIRS': True,
        'OPTIONS': {
            'context_processors': [
                'django.template.context_processors.debug',
                'django.template.context_processors.request',
                'django.contrib.auth.context_processors.auth',
                'django.contrib.messages.context_processors.messages',
                'investapp.context_processors.base_context',
            ],
        },
    },
]

WSGI_APPLICATION = 'investapp.wsgi.application'


# Database
# https://docs.djangoproject.com/en/3.1/ref/settings/#databases

DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.mysql',
        'NAME': env("MYSQL_NAME"),
        'USER': env("MYSQL_USER"),
        'PASSWORD': env("MYSQL_PASSWORD"),
        'HOST': env("MYSQL_HOST"),
        'PORT': env("MYSQL_PORT"),
    }
}

# Password validation
# https://docs.djangoproject.com/en/3.1/ref/settings/#auth-password-validators

AUTH_PASSWORD_VALIDATORS = [
    {
        'NAME': 'django.contrib.auth.password_validation.UserAttributeSimilarityValidator',
    },
    {
        'NAME': 'django.contrib.auth.password_validation.MinimumLengthValidator',
    },
    {
        'NAME': 'django.contrib.auth.password_validation.CommonPasswordValidator',
```

```
    },
    {
        'NAME': 'django.contrib.auth.password_validation.NumericPasswordValidator',
    },
]

AUTH_USER_MODEL = "profiles.UserProfile"

# Internationalization
# https://docs.djangoproject.com/en/3.1/topics/i18n/

LANGUAGE_CODE = 'es-es'

TIME_ZONE = 'Europe/Madrid'

USE_I18N = True

USE_L10N = True

USE_TZ = True


# Static files (CSS, JavaScript, Images)
# https://docs.djangoproject.com/en/3.1/howto/static-files/

STATIC_URL = '/static/'
STATICFILES_DIRS = [os.path.join(BASE_DIR, 'static')]
```

Empezando desde la parte superior del código:

- **SECRET_KEY**: Es una clave que se utiliza como firma criptográfica para muchas de las funcionalidades de seguridad de Django. Esta clave debe ser única y permanecer en todo momento secreta.
- **DEBUG**: Este valor booleano permite configurar el servidor de ejecución de Django en modo debug o producción. Cuando se encuentra en modo debug, si se produce un error, se muestran páginas detalladas de éste que incluyen muchos metadatos de la aplicación, como la configuración definida.
- **ALLOWED_HOSTS**: Esta lista permite definir los dominios desde los que la aplicación acepta peticiones.
- **INSTALLED_APPS**: Esta lista define todas las aplicaciones de Django que se utilizan en el proyecto. Además de las aplicaciones por defecto, para este proyecto se añaden: *profiles.apps.ProfilesConfig*, *markets.apps.MarketsConfig* y *chartjs*.
- **MIDDLEWARE**: En esta lista se definen los *middleware* que usa Django. Un middleware es un código que se ejecuta antes del procesamiento de petición/respuesta.

- **ROOT_URLCONF**: Define la ruta al fichero del proyecto en el que se configura el mapeado de URL.
- **TEMPLATES**: En este diccionario se define todo lo relacionado con las plantillas. En él se configura el motor que va a utilizar, los directorios en los que se van a colocar los ficheros HTML y los procesadores de contexto, que son los encargados de cargar dinámicamente las plantillas con los datos de contexto.
- **WSGI_APPLICATION**: Ruta al fichero en el que se define el punto de entrada de WSGI, aunque en este proyecto no se utiliza.
- **DATABASES**: En este diccionario se define la base de datos a utilizar.
- **AUTH_PASSWORD_VALIDATORS**: Lista de validadores para las contraseñas de usuarios.
- **AUTH_USER_MODEL**: Modelo que Django va a usar para representar a los usuarios. Es muy importante que este modelo se configure correctamente al empezar el proyecto, dado que crea muchas dependencias internas y por ello cambiarlo más adelante sería muy costoso.
- **Opciones de internacionalización y codificación**: Estas opciones sirven para configurar la zona horaria y las opciones de idioma de la aplicación.
- **Opciones de ficheros estáticos**: Estas dos opciones son las rutas a los directorios donde se colocan los ficheros estáticos (CSS, JS y HTML).

Continuando con la configuración a nivel de proyecto, queda mostrar cómo se enlazan todas las URL del proyecto. Esto se puede ver en el código del fichero *investapp/urls.py*:

```
from django.contrib import admin
from django.urls import path, include
from . import views

urlpatterns = [
    path('admin/', admin.site.urls),
    path('', views.index, name="main"),
    path('index', views.index, name="index"),
    path('risk_profile', views.risk_profile, name='risk_profile'),
    path('risk_profile_score', views.RiskProfileScore.as_view(), name="risk_profile_score"),
    path('profiles/', include('profiles.presentation.urls')),
    path('markets/', include('markets.presentation.urls'))
]
```

De este código cabe resaltar que el *test de perfil de riesgo* se ha colocado a nivel de proyecto y no en la aplicación de *profiles*. Se ha considerado que la aplicación solo debería tener responsabilidad de la propia gestión del usuario, y el test se puede realizar sin estar registrado como usuario de la web.

Para finalizar esta subsección, se va a exponer la implementación del *test de perfil de riesgo*.

El test consta de 20 preguntas con diferentes posibilidades de respuesta. Tanto las preguntas como las respuestas están basadas en un artículo científico con el título de *Financial risk tolerance revisited: the development of a risk assessment instrument* [56], en el que se formalizó cuáles son los aspectos relevantes para determinar la aversión al riesgo financiero de un inversor. Los resultados del test por otra parte, se han clasificado en tres posibilidades: BAJO, MODERADO y ALTO. Que son las clasificaciones usadas por la CNMV [11].

Su implementación consta de dos vistas, una para el formulario a realizar y otra para el resultado:

```
def risk_profile(request):

    if request.method == "GET":
        form = RiskProfileTestForm()
        return render(request, 'risk_profile_test.html', {'form': form})

    elif request.method == "POST":
        form = RiskProfileTestForm(data=request.POST)
        if form.is_valid():
            risk_profile_calculated = UserProfileService() \
                .calculate_risk_level(scores=tuple([int(e) for e in form.cleaned_data.values()]))
            request.session['risk_profile_calculated'] = risk_profile_calculated
            return redirect("risk_profile_score")


class RiskProfileScore(View):
    def get(self, request):
        risk_profile_calculated = request.session['risk_profile_calculated']
        return render(request, 'risk_profile_score.html',
        {'user_risk_lvl': risk_profile_calculated[0]['name'],
        'user_risk_score': risk_profile_calculated[1]})

    def post(self, request):
        if not request.user.is_authenticated:
            request.session['view_after_login'] = 'risk_profile_score'
            return redirect("user_login")

        user = UserProfile.objects.get_by_natural_key(username=request.user.username)
        user.update_risk_level(request.session['risk_profile_calculated'][0]['value'])
        return redirect("user_profile")
```

En la primera se puede ver el uso del Servicio de aplicación (UserProfileService) que se comentaba en el capítulo de arquitectura 5.4. A continuación en la página del resultado, si se envía una petición de tipo POST correspondiente al botón de guardar el resultado, se comprueba que el usuario haya iniciado sesión, de lo contrario se le envía a la página de login.

Respecto al formulario que se usa para implementar el test, se añade a continuación una muestra de su código que contiene todos los elementos necesarios para explicar la implementación:

```
from django import forms
from djchoices import DjangoChoices, ChoiceItem


class RiskProfileTestForm(forms.Form):
    class Question1(DjangoChoices):
        a = ChoiceItem(label="Un amante del riesgo.",
                       value="4")
        b = ChoiceItem(label="Alguien dispuesto a tomar riesgos después de una
        ↪  investigación adecuada.",
                       value="3")
        c = ChoiceItem(label="Una persona cautelosa.",
                       value="2")
        d = ChoiceItem(label="Una persona que evita el riesgo a toda costa.",
                       value="1")

    q1 = forms.ChoiceField(choices=Question1.choices, widget=forms.RadioSelect,
                           label="¿Cómo cree que le describiría su mejor amigo en cuanto a la
                           ↪  toma de riesgos?")
```

Para implementar las preguntas, se ha utilizado la biblioteca de *djchoices* [57], esta biblioteca proporciona una API sobre la implementación estándar de Django de las preguntas con múltiples posibilidades de respuesta. El uso de esta biblioteca se debe a que al haber muchas preguntas en el formulario, la implementación estándar resultaba en muchas líneas de código que dificultaban la lectura correcta de éste.

7.4.2. Aplicación de Profiles

Esta aplicación es la que ostenta las responsabilidades relacionadas con la gestión de los perfiles de usuario. Sus casos de uso son: Registro de usuario, inicio y desconexión de sesión, visualización de la página de perfil y modificación y borrado del perfil.

De estos casos de uso, cabe mencionar la implementación de la modificación y borrado. A continuación se muestra el código de sus URL y sus vistas:

```
    path("user/delete/<int:pk>/", views.DeleteUserProfile.as_view(),
                                  name="delete_user_profile"),
    path("user/update/<int:pk>/", views.UpdateUserProfile.as_view(),
                                  name="update_user_profile")

class UserProfilePermissionsMixin(AccessMixin):
    def dispatch(self, request, *args, **kwargs):
        if not self.user_has_permissions(request, kwargs['pk']):
            return self.handle_no_permission()
```

```
        return super(UserProfilePermissionsMixin, self).dispatch(request, *args, **kwargs)

    @staticmethod
    def user_has_permissions(request, requested_pk):
        return request.user.is_superuser or (request.user.pk == requested_pk)


class DeleteUserProfile(LoginRequiredMixin, UserProfilePermissionsMixin, DeleteView):
    model = UserProfile
    template_name = "user_profile_delete.html"
    success_url = "/"


class UpdateUserProfile(LoginRequiredMixin, UserProfilePermissionsMixin, UpdateView):
    model = UserProfile
    form_class = UserProfileUpdateForm
    template_name = "user_profile_update.html"
    success_url = "/profiles/user"

    def get_form(self, form_class=None):
        form = super(UpdateUserProfile, self).get_form()
        form.fields['fav_index'].label = 'Indice Preferido'
        return form

    def dispatch(self, request, *args, **kwargs):
        if request.method == "POST":
            if UserProfile.objects.filter(username=request.POST['username']).exists() and \
                    (UserProfile.objects.get_by_natural_key(username=request.POST['username']).pk
                    is not kwargs['pk']):
                return self.handle_no_permission()

        return super(UpdateUserProfile, self).dispatch(request, *args, **kwargs)
```

Django proporciona las clases DeleteView y UpdateView, que implementan el flujo de operaciones de estos casos de uso. Al crear una clase que hereda alguna de estas vistas, se puede personalizar para adecuarla al caso de uso de la aplicación. También se puede ver que se hereda de las clases LoginRequiredMixin y UserProfilePermissionsMixin. Para el LoginRequiredMixin, asegura que el usuario tiene sesión activa y válida. Para el UserProfilePermissionsMixin, permite que si el usuario es un administrador, pueda ejecutar estos casos de uso sobre perfiles de otros usuarios.

El modelo que utiliza esta aplicación es una extensión del básico que proporciona Django, añadiendo el perfil financiero y el índice bursátil favorito. Su implementación es la siguiente:

```
from django.contrib.auth.models import AbstractUser
from django.db import models

from investapp import settings as st
```

```
class UserProfile(AbstractUser):
    risk_level = models.IntegerField(default=st.RISK_PROFILE.BAJO.value)
    fav_index = models.CharField(default='^IBEX', max_length=20)

    def update_risk_level(self, risk_level: int):
        self.risk_level = risk_level
        self.save()

    def update_fav_index(self, index_ticker: str):
        self.fav_index = index_ticker
        self.save()
```

Por otro lado, el siguiente código muestra la implementación del Servicio de aplicación usado para gestionar el cálculo del resultado del test de perfil financiero:

```
from investapp import settings as st
from profiles.business.models import UserProfile


class UserProfileService:
    @staticmethod
    def calculate_risk_level(scores, username=None):
        """
        Calculates and updates the user profile (if the user is already authenticated)
        with the risk profile of the user from the answers of the test,
        associating the total score to one of the three risk levels established.
        NOTE: As the risk levels are fixed as business rule, the can be precalculated
              at the service start.
        :param scores: The scores of each question based on the answers of the user to the
               risk profile test.
        :type scores: tuple[int]
        :param username: Username of the user that has made the test if it was authenticated,
               otherwise is None.
        :type username: str
        :returns: The risk level calculated, with name and score.
        :rtype: tuple[str, float]
        """
        total_score = sum(scores)
        if total_score < st.GROUPS_LEFT_LIMITS.MODERADO.value:
            risk_level = st.RISK_PROFILE.BAJO
        elif total_score < st.GROUPS_LEFT_LIMITS.ALTO.value:
            risk_level = st.RISK_PROFILE.MODERADO
        else:
            risk_level = st.RISK_PROFILE.ALTO

        return {'name': risk_level.name, 'value': risk_level.value}, total_score
```

7.4.3. Aplicación de Markets

La aplicación de Markets es la encargada de gestionar los casos de uso referentes a la carga de datos financieros y al análisis de los Portfolios.

Las vistas de la aplicación son: listado de acciones, listado de mercados, detalle de una acción, detalle de un mercado y vista de Porfolio.
Antes de entrar en detalle en la implementación de las gráficas mostradas en las vistas de detalle. Es conveniente explicar la implementación de la conexión con la API del servicio de cálculos.

Para evitar que el servicio de aplicación quede acoplado al DAO, se ha implementado una factoría que es la encargada de la creación del mismo. De esta forma, el servicio recibe la implementación del DAO a través de ella. A continuación se muestra la implementación de éste:

```
class FincalcsDao(AbstractDao):
    def __init__(self):
        self.base_url = self.connect_to_data_source()

    def connect_to_data_source(self):
        return st.FINCALCS_HOST

    def get_all_stocks(self):
        url = self.base_url + st.FINCALCS_SYMBOLS_ENDPOINT + '/stocks'
        try:
            symbols = ujson.loads(HttpRequest(status_forcelist=[400, 404, 500]).get(url).content)
        except HttpRequestException as e:
            st.logger.exception(e)
            raise ExternalResourceError()
        else:
            return symbols

    def get_all_indexes(self):
        url = self.base_url + st.FINCALCS_SYMBOLS_ENDPOINT + '/indexes'
        try:
            indexes = ujson.loads(HttpRequest(status_forcelist=[400, 404, 500]).get(url).content)
        except HttpRequestException as e:
            st.logger.exception(e)
            raise ExternalResourceError()
        else:
            return indexes

    def get_symbol(self, ticker: str):
        url = self.base_url + st.FINCALCS_SYMBOLS_ENDPOINT + '/{}'.format(ticker)

        try:
            resp = HttpRequest(status_forcelist=[400, 500]).get(url)
            if resp.status_code == 404:
```

```
                raise SymbolNotFoundError()
            symbol = ujson.loads(resp.content)
        except HttpRequestException as e:
            st.logger.exception(e)
            raise ExternalResourceError()
        else:
            return symbol

    def get_portfolio(self, stocks, shares_per_stock, first_date_data, last_date_data):
        url = self.base_url + st.FINCALCS_PORTFOLIO_ENDPOINT
        try:
            tickers = ','.join(stocks)
            sps = ['{}:{}'.format(k,v) for k,v in shares_per_stock.items()]
            sps = ','.join(sps)
            body = {'tickers': tickers, 'sharesPerStock': sps, 'initial_date': first_date_data,
                    'end_date': last_date_data}
            resp = HttpRequest(status_forcelist=[400, 500]).post(url, body=body)
            if resp.status_code == 404:
                raise SymbolNotFoundError()
            portfolio = ujson.loads(resp.content)
        except HttpRequestException as e:
            st.logger.exception(e)
            raise ExternalResourceError()
        else:
            return portfolio
```

En cuanto a la implementación de los Portfolios. Cuando el usuario llega a la vista, lo primero que se le muestra es un formulario para crear el Portfolio. Este formulario permite seleccionar hasta 5 acciones distintas, y un número de posiciones (Acciones que se poseen) para cada una. En la figura 7.6, se muestra esta selección.

Figura 7.6: Selección de acción en el creador de Portfolio.

Este tipo de selección no existe por defecto en Django. Para poder crearla se ha tenido

que seguir la siguiente implementación:

```
class InputAndChoice(object):
    def __init__(self, quantity_val, choice_val=''):
        self.quantity_val = quantity_val
        self.choice_val = choice_val


class InputAndChoiceWidget(forms.widgets.MultiWidget):
    def __init__(self, *args, **kwargs):
        my_choices = kwargs.pop("choices")
        widgets = (
            forms.TextInput(attrs={'min': '0', 'type': 'number', 'value': '0'}),
            forms.widgets.Select(choices=my_choices)
        )
        super(InputAndChoiceWidget, self).__init__(widgets, *args, **kwargs)

    def decompress(self, value):
        if value:
            return [value.quantity_val, value.choice_val]
        return [None, None]


class InputAndChoiceField(forms.MultiValueField):
    def __init__(self, *args, **kwargs):
        # you could also use some fn to return the choices;
        # the point is, they get set dynamically
        my_choices = kwargs.pop("choices", [("default", "default choice")])
        fields = (
            forms.fields.CharField(),
            forms.fields.ChoiceField(choices=my_choices),
        )
        super(InputAndChoiceField, self).__init__(fields, *args, **kwargs)
        # here's where the choices get set:
        self.widget = InputAndChoiceWidget(choices=my_choices)

    def compress(self, data_list):
        if data_list:
            if int(data_list[0]) > 0 and isinstance(data_list[1], str):
                return int(data_list[0]), data_list[1]
        return None, None
```

Para terminar la subsección, queda hablar de la implementación de las gráficas en las vistas de detalle. Como se adelantó en el capítulo de tecnologías 4.4, se ha utilizado la librería de Chart.js. Esta librería no tiene por defecto soporte para las plantillas de Django. Por ello se ha utilizado la biblioteca de django-chartjs.

Esta biblioteca proporciona Clases que ofrecen métodos que proporcionan la interfaz que espera la biblioteca de Chartjs en el navegador. Por ejemplo, a continuación se adjunta

la implementación del gráfico de cierres para las acciones:

```
class symbol_detail_closures(BaseLineChartView):
    def get(self, request, *args, **kwargs):
        self._symbol = MarketsService().get_symbol(ticker=request.GET['ticker'])
        return super(symbol_detail_closures, self).get(request)

    def get_dataset_options(self, index, color):
        default_opt = super(symbol_detail_closures, self).get_dataset_options(index, color)
        default_opt['pointBorderWidth'] = '0.1'
        default_opt['pointRadius'] = '1.5'

        return default_opt

    def get_labels(self):
        return list(self._symbol['closures'].keys())

    def get_providers(self):
        return ['daily closes']

    def get_data(self):
        return [list(self._symbol['closures'].values())]
```

Los datos necesarios para generar la gráfica son los *providers*, que representan el eje X de la gráfica; las *labels*, para los valores del eje X y el *data*, los valores para el eje Y. Además en este caso se sobreescribe el método *get_dataset_options* para modificar la visualización.

En la plantilla correspondiente, se añade el siguiente script de JavaScript:

```
<script type="text/javascript">
    $.get('{% url "stock_closures" %}?ticker={{ticker}}', function (data) {
        var ctx = $("#closuresChart").get(0).getContext("2d");
        var options = {
            responsive: false,
            maintainAspectRatio: false,
            plugins: {
                zoom: {
                    zoom: {
                        enabled: true,
                        mode: 'x',
                    },
                    limits: {
                        y: {
                            min: 0
                        }
                    },
                }
            },
```

```
            scales: {
                xAxes: [{
                    stacked: false,
                    barPercentage: 10,
                    ticks: {

                    },
                }],
                yAxes: [{
                    stacked: true,
                    ticks: {

                    }
                }]
            }
        };
        new Chart(ctx, {
            type: 'line', data: data,
            options: options
        });
    });
```

En este código hay que mencionar que los datos se ponen disponibles a través de un *endpoint* específico, que se pasa como marcaje del lenguaje de plantillas de Django, ’% url ”stock_closures” %ticker=ticker’. De esta forma Django es capaz de traducirlo a la url específica, que es:

path('stocks/closures', views.stock_detail_closures.as_view(), name='stock_closures'),

7.5. Despliegue de contenedores con Docker Compose

Como se explicó en el capítulo de arquitectura en la sección 5.1. En el desarrollo de microservicios, uno de los impedimentos es disponer de un entorno de desarrollo con todos ellos. Para resolver este impedimento se ha implementado un despliegue de múltiples contenedores Docker usando Docker Compose.

Compose permite definir una serie de contenedores dependientes entre sí, a los que llama servicios, mediante un fichero YAML. A continuación, se va a exponer uno a uno la creación de los contenedores para cada microservicio.

7.5.1. RabbitMQ

En primer lugar, el fichero correspondiente al contenedor de RabbitMQ:

version: ”3.8”

```
services:
    rabbitmq:
        image: rabbitmq:3.8-management
        container_name: invest_system_rabbitmq
        environment:
            RABBITMQ_DEFAULT_USER: "finsystem"
            RABBITMQ_DEFAULT_PASS: "finsystemuserpass"
            RABBITMQ_DEFAULT_VHOST: "finsystem"
        ports:
            - "15672:15672"
            - "5672:5672"
        networks:
            - invest-system

networks:
    invest-system:
        external: true
```

El contenedor utilizado es el oficial, que incluye la consola de administración de RabbitMQ desde el navegador web. En la figura, se muestra el panel de administración de la cola consumidora perteneciente al servicio de cálculos.

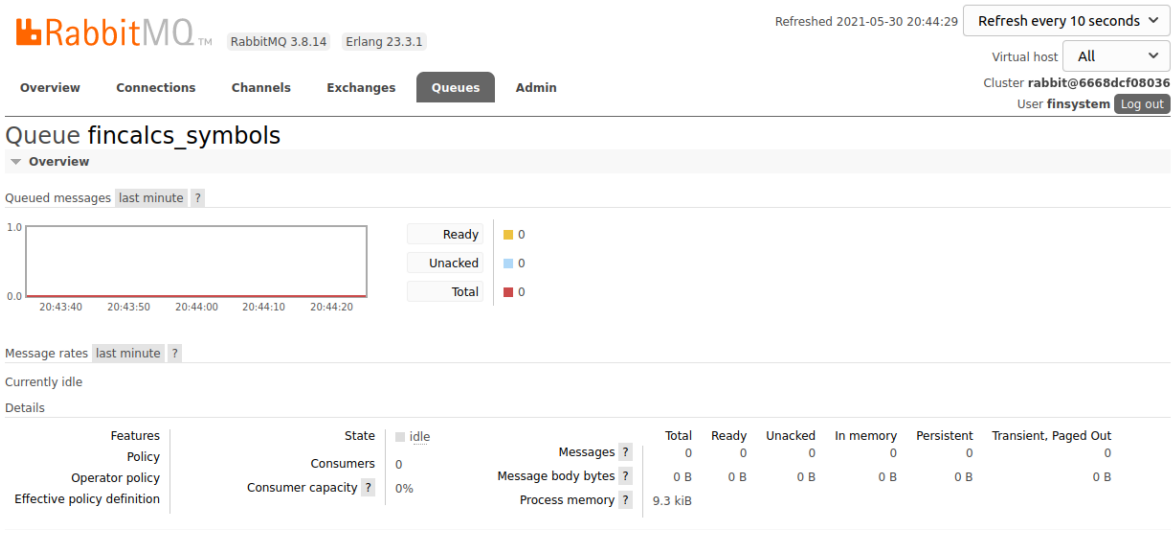


Figura 7.7: Ejemplo de Panel de administración de una cola de RabbitMQ.

Continuando con el fichero YAML. Para configurar el RabbitMQ, se le pasa un usuario con contraseña y un virtual host (vhost), estos hosts son espacios lógicos independientes que RabbitMQ utiliza para separar recursos. A continuación se define el mapeo de puertos contenedor-host y por último la red del contenedor. El concepto de red se explicará al final de la sección.

7.5.2. Servicio de recolección de datos

Para el microservicio de recolección de datos, su fichero es:

```
version: "3.8"

services:
  findata:
    container_name: findata_service
    build: .
    depends_on:
      mongodb:
        condition: service_started
    networks:
      - invest-system

  mongodb:
    image: mongo:4.4-bionic
    container_name: findata_mongodb
    environment:
      - MONGODB_USER=${MONGODB_USER}
      - MONGODB_PASS=${MONGODB_PASS}
    ports:
      - "27018:27017"
    volumes:
      - mongodata:/data/db
      - mongo-configdb:/data/configdb
    networks:
      - invest-system
    env_file: .env

volumes:
  mongodata:
  mongo-configdb:

networks:
  invest-system:
    external: true
```

Se definen dos servicios, el primero es el propio servicio de recolección. La sentencia build es el path al fichero dockerfile, que es el que define el contenedor del servicio. La sentencia depends_on, permite definir dependencias con otros contenedores, en este caso se indica que antes de ejecutar el contenedor de findata, debe esperar a que el servicio de MongoDB esté levantado. Esto solo garantiza que los contenedores estén en ejecución, pero no que los servicios que despliegan en su interior lo estén.

Después, está el servicio de MongoDB. De este apartado hay que destacar el uso de volúmenes para persistir los datos de mongo entre reinicios del contenedor. Por último, se vuelve a definir el contenedor de RabbitMQ. Esto es necesario porque si el servicio no existe previamente, se debe levantar aquí.

En cuanto al dockerfile de Findata, se muestra a continuación:

```
FROM python:3.9-buster

RUN apt-get -y update
RUN apt-get -y install git

RUN useradd --create-home findatauser

WORKDIR /home/findatauser/findata

COPY requirements.txt .
RUN pip install -r requirements.txt

COPY main.py  .
COPY .env .
RUN touch findata.log findata_errors.log
COPY src ./src

RUN chown -R findatauser .
RUN chmod -R 700 .

USER findatauser

ENTRYPOINT ["python", "main.py"]
```

A la hora de construir un fichero Dockerfile es importante tener en cuenta el orden en el que se colocan las entradas. En la construcción de un contenedor, Docker utiliza un sistema de capas. Esto significa que a cada paso, genera un contenedor que agrega la entrada actual y todas las anteriores. De esta forma se construye un sistema de caché que permite reutilizar la construcción de contenedores siempre que los cambios sean iguales. Por ello, para aprovechar al máximo esta característica de Docker, se puede poner por ejemplo la instalación de las dependencias del proyecto antes de la copia de los ficheros de la máquina host al contenedor. Ya que es mucho más común que cambien los ficheros de código a que cambien las dependencias.

Tras la copia de los ficheros, se cambian los permisos de estos a nivel de sistema operativo. Un contenedor Docker utiliza el usuario Root por defecto, por seguridad, es recomendable rebajar los permisos del código a ejecutar.

7.5.3. Servicio de cálculos financieros

A continuación, se muestra el fichero YAML del servicio de cálculos financieros:

```
version: "3.8"

services:
  mongodb:
    image: mongo:4.4-bionic
    container_name: fincalcs_mongodb
```

```
    environment:
      - MONGODB_USER=${MONGODB_USER}
      - MONGODB_PASS=${MONGODB_PASS}
    ports:
      - "27017:27017"
    volumes:
      - mongodata:/data/db
      - mongo-configdb:/data/configdb
    networks:
      - invest-system
  fincalcs:
    container_name: fincalcs_service
    build: .
    ports:
      - "8001:8001"
    depends_on:
      mongodb:
        condition: service_started
    networks:
      - invest-system
    env_file: .env

volumes:
  mongodata:
  mongo-configdb:

networks:
  invest-system:\textbf{}
```

Al igual que en el anterior microservicio, este tiene como dependencias un MongoDB. En cuanto al dockerfile, no tiene diferencias significativas con el anterior.

7.5.4. Aplicación Web

Para terminar el capítulo, queda mostrar la implementación de los contenedores para la aplicación web.

El fichero YAMl es el siguiente:

```
version: "3.8"

services:
  db:
    image: mysql:8.0.24
    container_name: investapp_db
    command: --default-authentication-plugin=mysql_native_password
    restart: always
    environment:
```

```
      - MYSQL_ROOT_PASSWORD=${MYSQL_ROOT_PASSWORD}
      - MYSQL_DATABASE=${MYSQL_DATABASE}
      - MYSQL_USER=${MYSQL_USER}
      - MYSQL_PASSWORD=${MYSQL_PASSWORD}
    ports:
      - "3306:3306"
    expose:
      - "3306"
    networks:
      - invest-system
  web:
    build: .
    entrypoint: ./entrypoint.sh
    volumes:
      - .:/investApp
    ports:
      - "8000:8000"
    depends_on:
      db:
        condition: service_started
    networks:
      - invest-system
    env_file: .env

networks:
  invest-system:
    external: true
```

De este despliegue, cabe destacar que se utiliza un pequeño script de tipo bash para inicializar el servidor de Django. El script aplicará las migraciones de Django y después lanzará el servidor con la IP 0.0.0.0, esto es importante, ya que de otro modo el contenedor no será accesible desde el exterior.

En cuanto al dockerfile del servicio web, no tiene diferencias significativas respecto al mostrado en la subsección 7.5.2.

7.5.5. Configuración de red para los contenedores

Para cerrar el capítulo, queda hablar de como se deben configurar los contenedores para que puedan comunicarse entre sí.

En docker existen varios tipos de adaptador de red:

- **Bridge**: Las redes de este tipo, se utilizan se usan cuando se tienen varios contenedores que necesitan comunicarse entre sí.
- **Host**: Este tipo de redes eliminan el aislamiento entre el contenedor y el Host permitiendo usar su red directamente.

- **Overlay**: Estas redes se usan para dar soporte a la herramienta de orquestación Docker Swarm. Sirve para conectar múltiples demonios de Docker entre sí, permitiendo así que los servicios de Swarm se comuniquen.
- **Macvlan**: Estas redes permiten asignar una dirección MAC al contenedor, permitiendo así que la red del Host lo identifique como un dispositivo físico. Este tipo de redes se suelen usar cuando la aplicación que se ejecuta dentro del contenedor necesita conexión directa con la red física.

Para este proyecto se ha utilizado el adaptador de tipo Bridge, ya que es el que mejor encaja en el caso de uso. En todos los ficheros de tipo compose, se puede ver que los contenedores conectan con una red llamada *invest-system*, esta red tiene la sentencia *external* con valor true. Esto significa que la red es preexistente al contenedor. En este caso, antes de crear cualquier contenedor, se ha creado la red para que todos los servicios puedan usarla. Para ello, se ha ejecutado el comando: *docker network create -d bridge invest-system.*

Capítulo 8

Conclusiones y trabajo futuro

Al comienzo de esta memoria se plantearon en la sección 1.3 una serie de objetivos que el trabajo debía cumplir para considerarse completo. En este capítulo se va a hacer un repaso de los resultados obtenidos en relación a la consecución de estos objetivos.

- **Desarrollo de un servicio para recoger datos financieros:** El servicio que se ha desarrollado permite la recolección de datos financieros de forma periódica a partir de diferentes fuentes de datos. Su arquitectura permite que se añadan nuevas fuentes de datos o que se modifiquen las ya existentes sin afectar apenas al resto del sistema, gracias al propio diseño y al versionado de éste.

  Como conclusión a este objetivo, se puede decir que el uso del diseño hexagonal, así como el desacoplamiento de este servicio del resto del sistema gracias al uso de RabbitMQ, ha resultado ser una estrategia muy eficaz en el desarrollo de este tipo de servicios fundamentales para un sistema de esta tipología.

- **Desarrollo de un servicio de cálculos financieros:** Este objetivo se constituía como el núcleo del trabajo, dado que es en el servicio desarrollado para completarlo, donde se producen todos los cálculos relacionados al análisis de riesgo de una cartera de inversión.

  Se puede afirmar que el objetivo se ha cumplido de forma satisfactoria. Los cálculos que el servicio ofrece generan una imagen sólida de la situación de una cartera de inversión en cuanto a su riesgo. Además, el servicio permite al usuario plantear diferentes escenarios de inversión de una forma versátil y basada en el histórico real de los símbolos.

- **Desarrollo de una aplicación web:** Para satisfacer este objetivo, se ha llevado a cabo el desarrollo de una aplicación web que en su resultado final, permite a los usuarios tanto la consulta de información de los mercados financieros como la prueba de diferentes escenarios para sus carteras de inversión. La elección de tecnologías ha resultado muy acertada a la hora de desarrollar esta aplicación. En base al resultado conseguido se puede decir que el objetivo planteado se ha logrado.

- **Reutilización y escalabilidad** Este último objetivo planteaba la necesidad de que todos los elementos del sistema fuesen reutilizables y escalables. Esta idea ha estado

presente durante el desarrollo de todos los elementos del sistema y ha determinado en gran parte la elección de la arquitectura del sistema, así como el diseño de cada servicio. En base a los resultados obtenidos se puede afirmar que el objetivo se ha cumplido de forma satisfactoria.

- **Conclusión final:** Además de lo expuesto anteriormente, una conclusión derivada de este trabajo es que el uso de arquitecturas de microservicios, resulta especialmente eficaz en un dominio tan extenso como es el sector financiero. Todos los datos y cálculos pueden ser utilizados para dar diferentes soluciones y aportar valor al usuario. Es aquí, donde la capacidad de la arquitectura de microservicios da una especial agilidad al sistema, permitiendo interconectar los servicios en diferentes formas para dar soporte a todas las necesidades que vayan surgiendo.

  Como conclusión final, se puede establecer que todos los objetivos planteados se han cumplido correctamente. El sistema resultante de este trabajo constituye una plataforma que aporta valor al usuario final y es una base sólida para llevar a cabo ampliaciones y mejoras en el futuro.

Para finalizar el capítulo, se van a exponer algunas mejoras y ampliaciones del sistema que podrían constituir un conjunto de líneas de trabajo futuro:

- **Ampliación de las fuentes de datos**: Una mejora clara para el sistema sería añadir más fuentes de datos, permitiendo mejorar así tanto la cantidad como la tolerancia a errores que puedan tener los proveedores de datos.
- **Recolección de más instrumentos financieros**: Actualmente el sistema sólo permite operar con acciones e índices bursátiles, no obstante, ha sido diseñado para añadir más tipos de instrumentos financieros como fondos de inversión o bonos de deuda.
- **Desarrollo de un sistema de optimización de carteras de inversión**: Actualmente el sistema permite al usuario analizar una cartera de inversión, pero la optimización de ésta debe hacerse de forma manual. Una funcionalidad que aportaría un valor añadido, sería ofrecerle al usuario la optimización de la cartera.
- **Agregación de noticias financieras**: Agregar noticias relacionadas con los mercados podría permitir al usuario tener mayor información contextual de los movimientos que se muestran en el histórico de cada símbolo.

Capítulo 9

Conclusions and future work

At the beginning of this report, a series of objectives were established in the section 2.3 that the work had to meet in order to be considered complete. This chapter will review the results obtained in relation to the achievement of these objectives.

- **Development of a service to collect financial data:** The service that has been developed allows the collection of financial data periodically from different data sources. Its architecture allows new data sources to be added or existing ones to be modified without affecting the rest of the system, thanks to its own design and versioning.

  As a conclusion to this objective, it can be said that the use of the hexagonal design, as well as the decoupling of this service from the rest of the system thanks to the use of RabbitMQ, has proved to be a very effective strategy in the development of this type of fundamental services for a system of this type.

- **Development of a financial calculation service:** This objective was constituted as the core of the work, since it is in the service developed to complete it, where all the calculations related to the risk analysis of an investment portfolio take place.

  It can be stated that the objective has been satisfactorily met. The calculations provided by the service generate a solid picture of the situation of an investment portfolio in terms of risk. In addition, the service allows the user to propose different investment scenarios in a versatile way and based on the actual history of the symbols.

- **Web application development:** To meet this objective, a web application has been developed which, in its final result, allows users both to consult information on financial markets and to test different scenarios for their investment portfolios. The choice of technologies has been very successful in the development of this application. Based on the results obtained, it can be said that the objective has been achieved.

- **Reusability and scalability** This last objective required that all the elements of the system be reusable and scalable. This idea has been present during the development of all the elements of the system and has largely determined the choice of the system architecture, as well as the design of each service. Based on the results obtained, it can be affirmed that the objective has been satisfactorily achieved.

- **Final conclusion:** In addition to the above, a conclusion derived from this work is that the use of microservices architectures is especially effective in such a large domain as the financial sector. All the data and calculations can be used to provide different solutions and bring value to the user. It is here, where the capacity of the microservices architecture gives a special agility to the system, allowing to interconnect the services in different ways to support all the needs that arise.

  As a final conclusion, it can be stated that all the objectives have been successfully met. The system resulting from this work constitutes a platform that provides value to the end user and is a solid base for future extensions and improvements.

To conclude the chapter, some improvements and extensions to the system that could constitute a set of lines of future work will be presented:

- **Expansion of data sources**: A clear improvement to the system would be to add more data sources, thus improving both the quantity and the error tolerance of data providers.
- **Collection of more financial instruments**: Currently the system only allows trading stocks and stock indexes, however, it has been designed to add more types of financial instruments such as mutual funds or debt bonds.
- **Development of an investment portfolio optimization system**: Currently the system allows the user to analyze an investment portfolio, but the optimization of the portfolio must be done manually. A functionality that would add value would be to offer the user the optimization of the portfolio.
- **Aggregation of financial news**: Adding news related to the markets could allow the user to have more contextual information of the movements shown in the history of each symbol.

Anexo A

Guía de uso de la aplicación web

En este anexo se va a mostrar una guía básica de uso de la aplicación web.

En primer lugar, cuando el usuario accede a la página obtiene la siguiente pantalla (Figura A.1):

Figura A.1: Página principal de InvestApp.

En esta página, en la parte superior derecha, se pueden ver dos opciones: Registrarse y Conectarse. Asumiendo que el usuario no se ha registrado previamente, se pulsa sobre Registrarse. A continuación se visualizará la vista de registro en las figuras A.2 y A.3.

Figura A.2: Formulario de registro de usuario.

Figura A.3: Formulario de registro de usuario.

Una vez introducidos los datos, cumpliendo con las restricciones de cada campo, se pulsa el botón enviar. Si el registro se completa correctamente, se mostrará la página de perfil de usuario que se puede ver en la figura A.4. De lo contrario, se volvería al formulario y se observaría un informe de que error se ha producido.

Figura A.4: Página de perfil de usuario.

En la página de perfil, se puede observar diferente información. A la izquierda, la información de usuario, con botones para modificar o eliminar el perfil. A la derecha, el resumen del índice favorito del usuario, que hasta que el usuario lo modifique, será el Ibex35. Por último, abajo se muestra el perfil de riesgo del usuario junto a una explicación de lo que éste representa y un botón para acceder al test.

A continuación, se mostrará el formulario (Figura A.5) para modificar el perfil que aparece como resultado de pulsar en el botón azul del panel de información de usuario. Para modificar un campo, el usuario simplemente debe reescribirlo. En el caso del índice, se mostrará un desplegable con la lista de todos los índices bursátiles disponibles en el sistema.

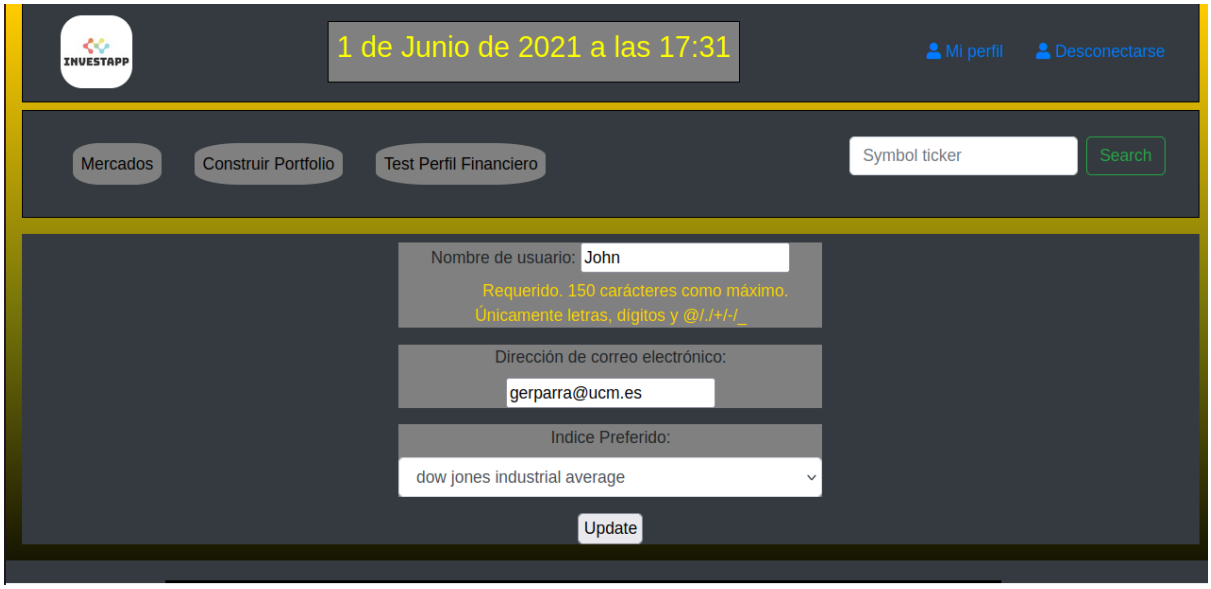


Figura A.5: Formulario de modificación de usuario.

Una vez introducido los nuevos datos, se volverá a la página de perfil (Figura A.4).

Tanto si se pulsa en el botón *cambiar* del perfil de riesgo de la página de perfil, como si se pulsa en el menú de navegación el botón *Test Perfil Financiero*. Se mostrará la vista con el formulario para rellenar el test, en la figura A.6 se muestra el principio de este formulario. Una vez seleccionadas todas las respuestas se pulsará en el botón enviar al final del formulario.

Figura A.6: Test de perfil de riesgo.

Tras enviar las respuestas del formulario, se calcula el perfil y se muestra el resultado en la siguiente página (Figura A.7):

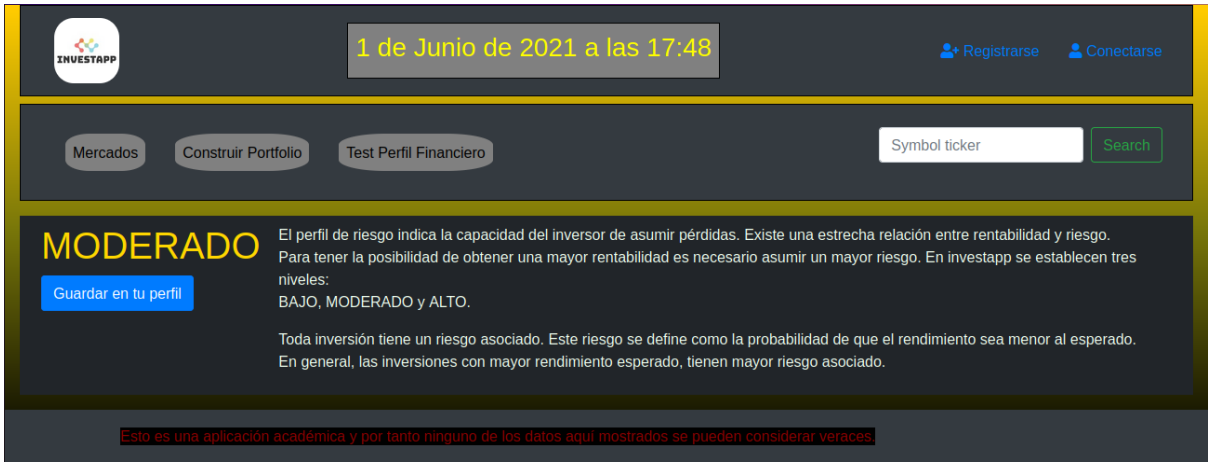


Figura A.7: Resultado del test de perfil financiero.

Si se pulsa en *Guardar en tu perfil*, se modificará el perfil de riesgo del usuario. En caso de que no se haya iniciado sesión previamente, se mostrará la página de inicio de sesión (Figura A.8):

Figura A.8: Inicio de sesión.

Por otro lado, si se pulsa en el menú de navegación la opción *Mercados*, se mostrará el listado de mercados (Figura A.9) disponibles:

Figura A.9: Listado de mercados.

Si por ejemplo se selecciona el mercado Euronext100, se visualizará su vista de detalle. Esta vista desde la parte superior a la inferior contiene:

- Un panel con el resumen del mercado que muestra su ticker, un enlace para visualizar las acciones que lo componen, la primera y última fecha del histórico y la tasa de crecimiento anual compuesto (CAGR) a 3 y 5 años. (Figura A.10).
- La gráfica con los cierres diarios del mercado. En el eje de abscisas se muestra el día y en el de ordenadas el valor del cierre. (Figura A.11).
- La gráfica con los retornos diarios positivos, en verde, y negativos, en rojo, del mercado. En el eje de abscisas se muestra el día y en el de ordenadas el valor por cien del retorno. (Figura A.12).

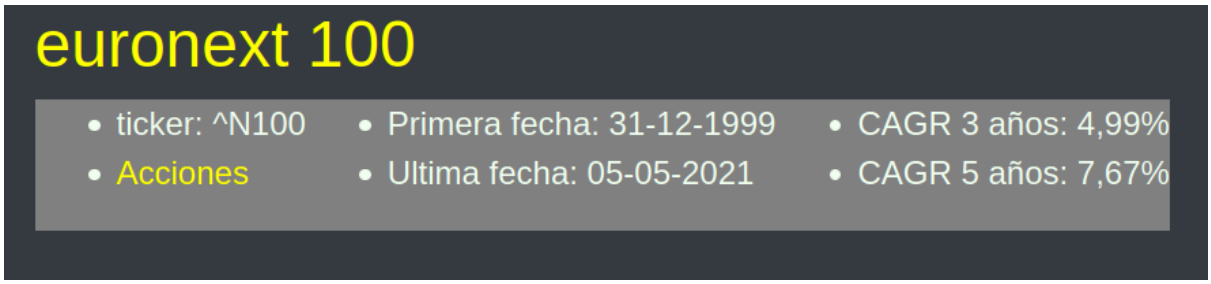


Figura A.10: Detalles del mercado Euronext100.

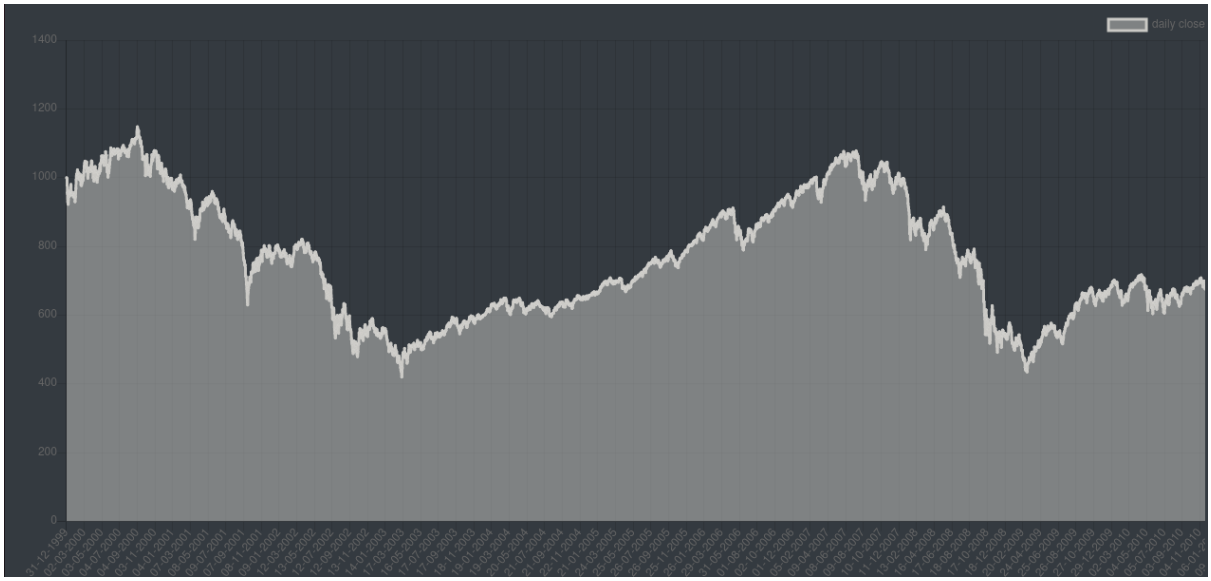


Figura A.11: Cierres del mercado Euronext100.

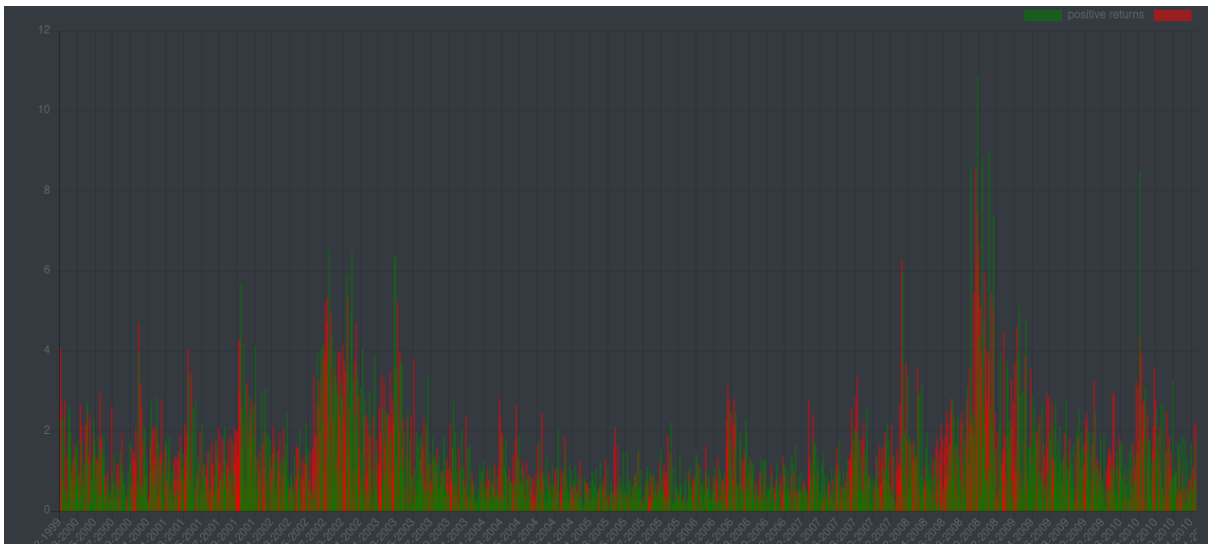


Figura A.12: Retornos del mercado Euronext100

Si se pulsa en el enlace *acciones*, que se puede ver en amarillo en la figura A.10, se mostrará el listado de acciones (Figura A.13) que pertenecen a este mercado:

Figura A.13: Listado de acciones del Euronext100

Al seleccionar cualquiera de las acciones, se muestra su vista de detalle. Esta vista consiste en:

- Un panel con el resumen de la acción que contiene: el ticker, isin y mercado, la primera y última fecha del histórico y la tasa de crecimiento anual compuesto (CAGR) a 3 y 5 años. (Figura A.14).
- Una gráfica con los cierres diarios y los dividendos. En el eje de abscisas se muestra el día y en el de ordenadas el valor del cierre y cuando corresponde, el del dividendo. (Figura A.15).
- La gráfica con los retornos diarios positivos, en verde, y negativos, en rojo, del mercado. En el eje de abscisas se muestra el día y en el de ordenadas el valor por cien del retorno. (Figura A.16).

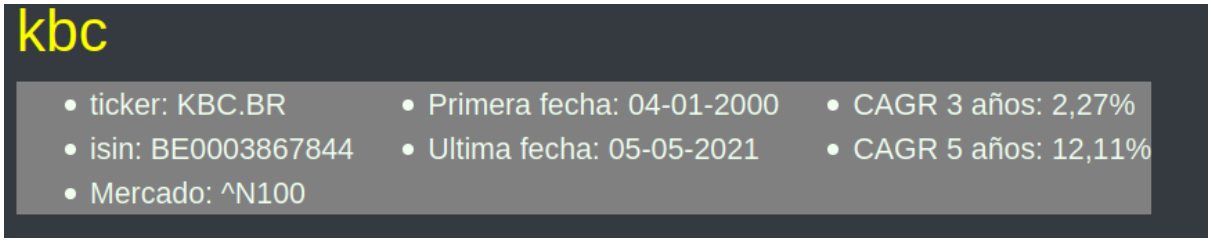


Figura A.14: Detalles de la acción kbc.

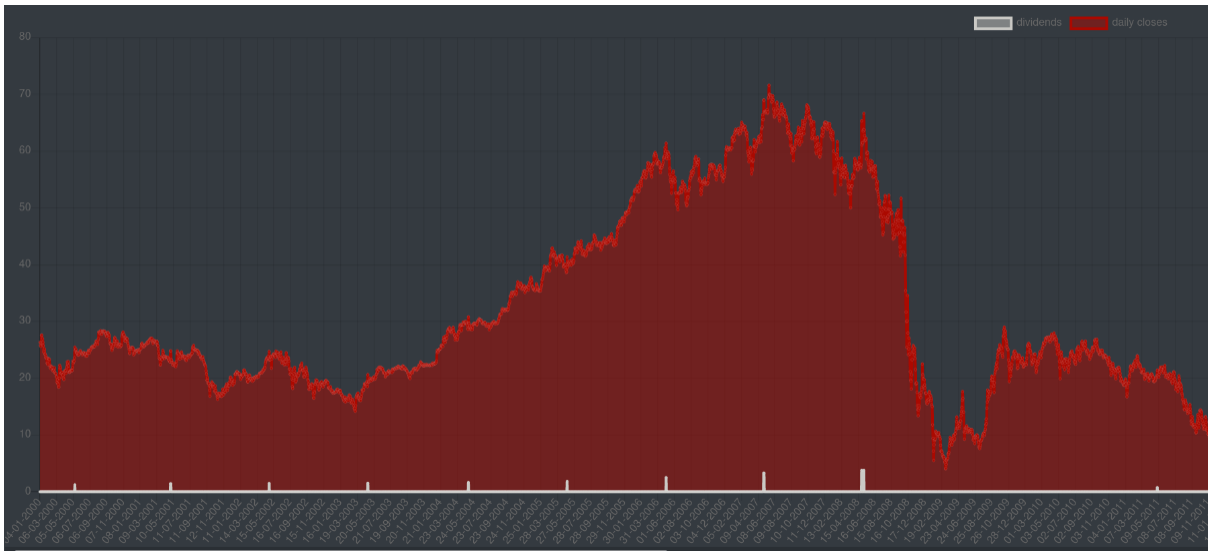


Figura A.15: Cierres diarios de la acción kbc.

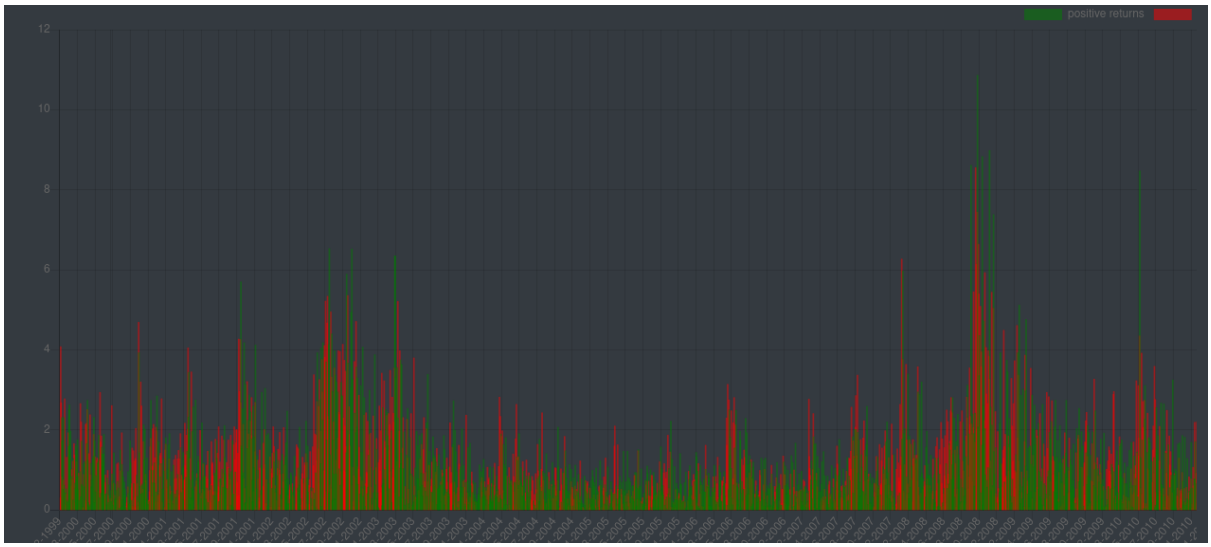


Figura A.16: Retornos de la acción kbc.

Para acabar esta guía de usuario queda mostrar la creación del análisis de una cartera de inversión.

En el menú de navegación se debe pulsar en la opción *Contruir Portfolio*. Esto nos llevará a la vista de construcción (Figura A.18), que consiste en una selección de hasta 5 acciones distintas a elegir entre todas las acciones disponibles del sistema.

Para seleccionar una acción, se debe elegir cuál se quiere en el desplegable (Figura A.17). Este desplegable mostrará todas las acciones del sistema ordenadas alfabéticamente por el ticker de su mercado, con el siguiente formato: *(ticker) Nombre de la acción - Ticker del Mercado*. Una vez seleccionada la acción, se debe colocar un número de posiciones mayor que 0. Si no se desea seleccionar una acción, se dejará el campo a 0.

Figura A.17: Selección de acción en el creador de Portfolio.

Una vez seleccionadas las acciones, se introduce la fecha de inicio y de final. La fecha de inicio debe ser anterior a la final y la fecha final debe ser como mucho la de ayer. La aplicación será capaz de seleccionar el mayor número de datos disponible dentro de la ventana temporal dada.

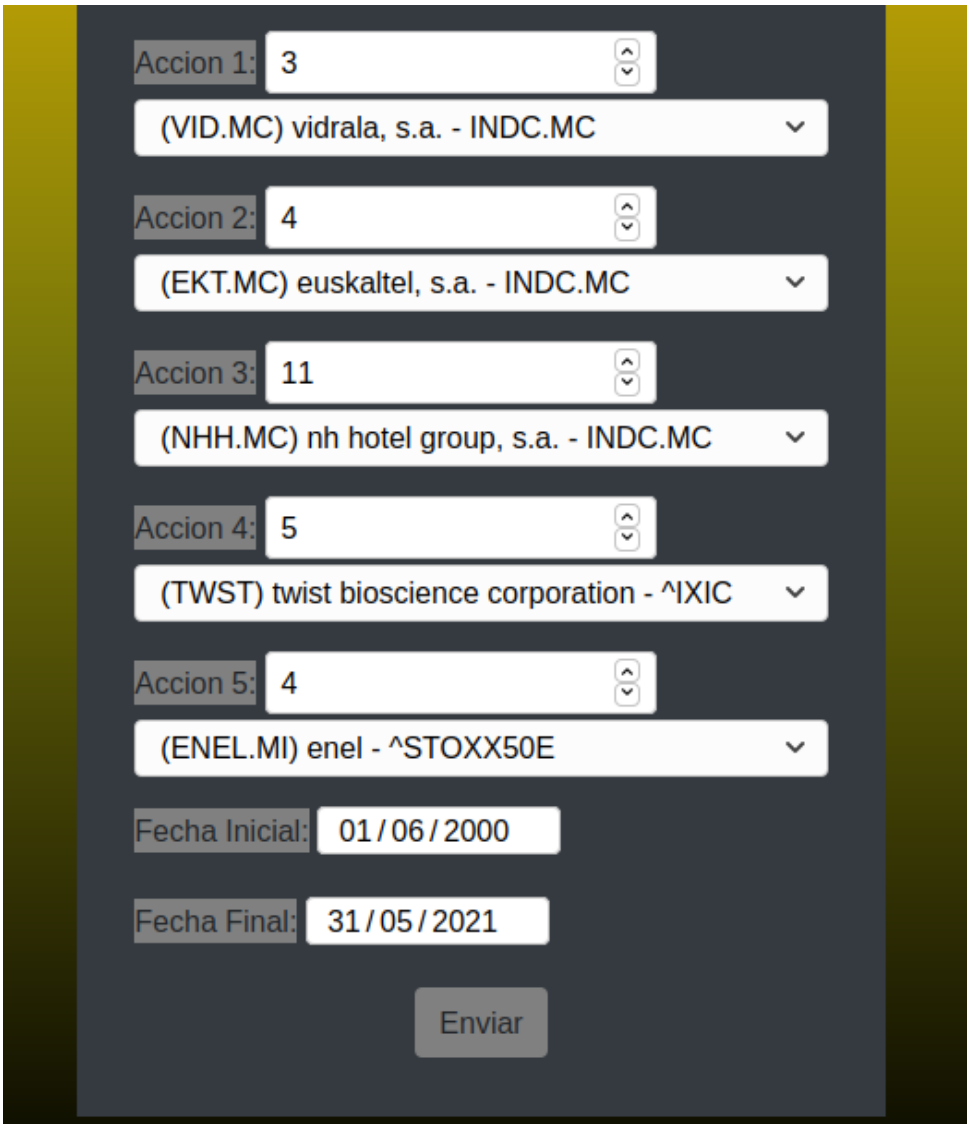


Figura A.18: Construcción de un Portfolio.

El resultado del análisis (Figuras A.19, A.20 y A.21) de este Portfolio se mostrará en una sola vista que consiste en:

- Un panel con la información del análisis. De izquierda a derecha, la primera columna muestra el ticker de cada acción, cada ticker es un enlace al detalle de la acción. Debajo de cada ticker, se muestra el porcentaje de la cartera que representa la acción. La segunda columna muestra la primera y última fecha que componen la ventana temporal del histórico del análisis. La tercera columna muestra los cálculos correspondientes al análisis del Portfolio a excepción del sortino que se mira en la cuarta columna. El sortino ratio se calcula utilizando como *benchmark* cada uno de los índices bursátiles a los que las acciones de la cartera pertenecen. Para cada benchmark se muestra el ticker del índice, con un enlace a su vista de detalle y el valor del ratio. (Figura A.19).
- La gráfica con los retornos diarios positivos, en verde, y negativos, en rojo, de la cartera. En el eje de abscisas se muestra el día y en el de ordenadas el valor por cien del retorno. (Figura A.20).

- La gráfica con la volatilidad diaria de la cartera. En el eje de abscisas se muestra el día y en el de ordenadas el valor. (Figura A.21).

Figura A.19: Resultado del análisis del Portfolio.

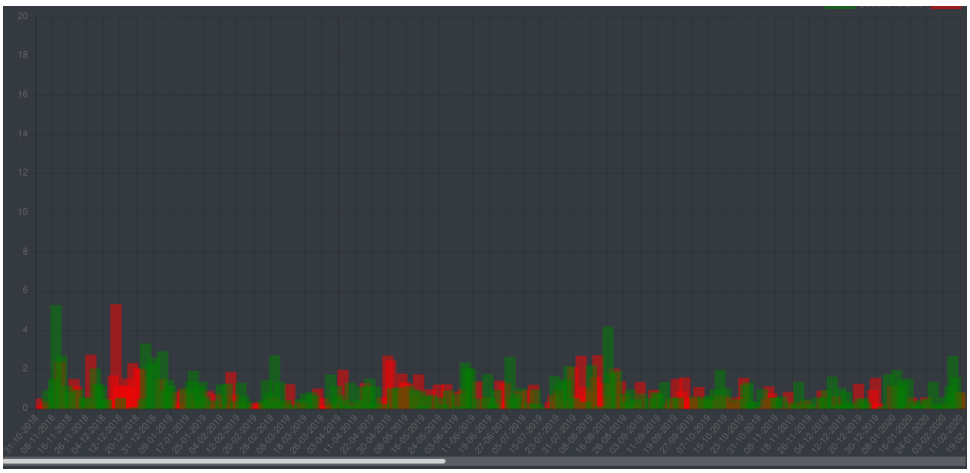


Figura A.20: Retornos del Portfolio.

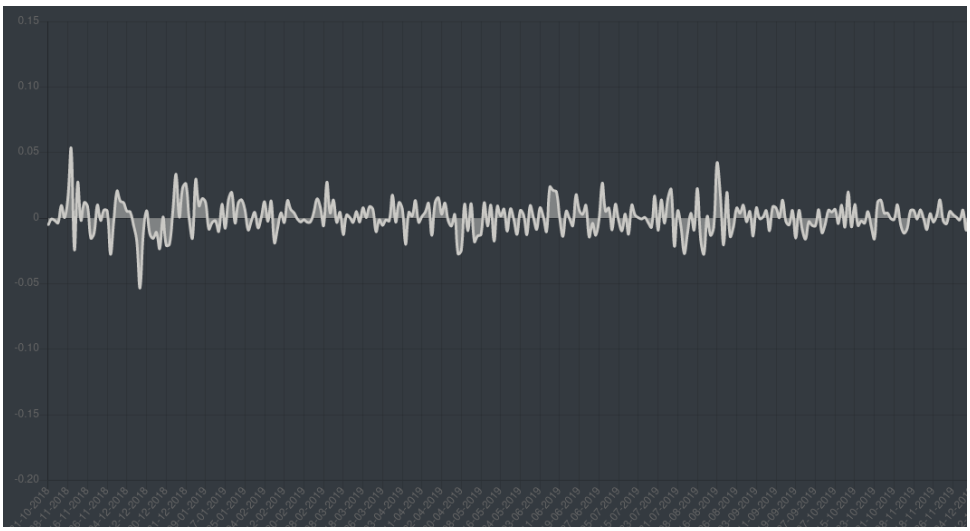


Figura A.21: Volatilidad del Portfolio.

Anexo B

Guía del Test de perfil financiero

Este anexo sirve como guía para entender el test de perfil financiero de la aplicación web.

Este test va a permitir al usuario conocer una estimación de su perfil inversor. El test esta basado en el artículo *Financial risk tolerance revisited: the development of a risk assessment instrument* [56]. Este artículo formalizó cuáles son los aspectos relevantes para determinar la aversión al riesgo financiero de un inversor. En él se propone un test en el que el resultado tenga en cuenta las múltiples dimensiones del riesgo. De esta forma el resultado final es una estimación consistente del perfil inversor del usuario.

El test consta de 20 preguntas con diferentes posibilidades de respuesta. Los resultados del test por otra parte, se han clasificado en tres posibilidades: *bajo*, *moderado* y *alto*. Que son las clasificaciones usadas por la CNMV [11].

Para acceder al test, como se explicaba en el anexo A, se puede hacer desde el menú de navegación en la opción *Test Perfil Financiero* o desde el perfil de usuario en el apartado del perfil financiero, pulsando el botón azul de *cambiar*. (Figura A.4)

Una vez se ha accedido, se mostrará el formulario con todas las preguntas y un botón de enviar al final de éste. En las figuras A.2 y A.3 se muestra la parte superior y la inferior de este formulario. A continuación se va a listar todas las preguntas y sus respuestas, la puntuación asignada a cada respuesta y, por último, la relación entre las preguntas y las dimensiones del riesgo.

1. ¿Cómo cree que le describiría su mejor amigo en cuanto a la toma de riesgos?
   a) Un amante del riesgo.
   b) Alguien dispuesto a tomar riesgos después de una investigación adecuada.
   c) Una persona cautelosa.
   d) Una persona que evita el riesgo a toda costa.
2. Está participando en un concurso y debe escoger una de las siguientes opciones, ¿Cúal escogería?
   a) Ganar 1.000€.

b) Una probabilidad del 50 % de ganar 5.000€.

c) Una probabilidad del 25 % de ganar 10.000€.

d) Una probabilidad del 5 % de ganar 100.000€.

3. Acaba de terminar de ahorrar para unas vacaciones de lujo. A tres semanas antes de irse, pierde su empleo, ¿Qué haría?

   a) Cancelar las vacaciones.

   b) Reducir los planes para recortar gastos.

   c) Seguir con el plan, ya que utilizará ese tiempo para preparar su búsqueda de un nuevo empleo.

   d) Extender sus vacaciones, ya que estas podrían ser sus últimas vacaciones a lo grande.

4. Considere la siguiente afirmación, 'Es difícil que deje pasar una oferta:

   a) Cierto.

   b) Podría ser.

   c) En absoluto.

5. Recibe 20.000€ para dedicarlos a inversión, ¿qué haría?

   a) Depositarlos en una cuenta de ahorro o invertir en deuda de corto plazo (Money Market).

   b) Invertir en bonos de alta calidad o fondos de inversión de bonos.

   c) Invertir en acciones o fondos de inversión de acciones.

6. A la hora de invertir en acciones o fondos de acciones, ¿Cómo de cómodo se siente?

   a) Me genera incomodidad.

   b) Me siento más o menos cómodo.

   c) Me siento bastante confiado.

7. ¿Cuál de las siguientes situaciones le daría más alegría?

   a) Ganar 50.000€ en un concurso.

   b) Heredar 50.000€ de un pariente rico.

   c) Ganar 50.000€ habiendo arriesgado 1.000€ en el mercado de acciones.

   d) Cualquiera de las anteriores está bien, en cualquier caso, ha ganado 50.000€.

8. ¿Qué idea le evoca la palabra 'riesgo'?

   a) Pérdidas.

   b) Incertidumbre.

   c) Oportunidad.

   d) Emoción.

9. Ha heredado una propiedad libre de cargas con un valor de 200.000€. Usted espera que su valor aumente más rápido que la inflación, sin embargo, necesita arreglos. Si la alquila ahora, recibirá 600€ mensuales, pero si la arregla antes, serán 800€. Para financiar los arreglos, tendría que hipotecarla, ¿que haría?

   a) Vender la casa.

   b) Alquilar sin reparación.

   c) Arreglarla y alquilarla después.

10. ¿Cuál de las dos afirmaciones es más importante para usted?

    a) Es más importante asegurar mi dinero evitando pérdidas o robos.

    b) Es más importante estar protegido frente a la inflación.

11. Consigue empleo en una 'start-up' de rápido crecimiento. Tras el primer año se le ofrecen las siguientes opciones, ¿cuál escogería?

    a) Garantizar el empleo por 5 años.

    b) Un bonus de 25.000€.

    c) Acciones de la compañía por valor de 25.000€ con la intención de venderlas a mayor valor más tarde.

12. Algunos expertos predicen que el valor de los bonos podría caer mientras que el valor de activos como el oro o los bienes inmobiliarios (bienes tangibles), incrementarán su valor. Por otro lado, la mayoría de los expertos coinciden en que los bonos estatales son bastante seguros. La mayoría de su cartera está formada por bonos estatales. ¿Qué haría?

    a) Mantener los bonos.

    b) Vender los bonos, invertir la mitad de las ganancias en deuda a corto plazo (Money Market), y la otra mitad en bienes tangibles.

    c) Vender los bonos y utilizar todas las ganancias en bienes tangibles.

    d) Vender los bonos y utilizar todas las ganancias además de dinero adicional en bienes tangibles.

13. Está planeando comprar una casa en las próximas semanas, ¿qué estrategia seguiría?

    a) Comprar una casa que le permita pagar las mensualidades de la hipoteca de forma cómoda.

    b) Comprar la casa que realmente le gusta, aunque tenga que apretarse el cinturón.

    c) Comprar la casa más cara que pueda permitirse.

    d) Pedir algo de dinero a amigos y familia para poder obtener una hipoteca mayor.

14. ¿Cuál de las siguientes 4 inversiones prefiere?

a) 200€ de ganancias en el mejor caso, ninguna pérdida en el peor caso.

b) 800€ de ganancias en el mejor caso, 200€ de pérdidas en el peor caso.

c) 2.600€ de ganancias en el mejor caso, 800€ de pérdidas en el peor caso.

d) 4.800€ de ganancias en el mejor caso, 2.400€ de pérdidas en el peor caso.

15. Esta solicitando una hipoteca para una casa en la que planea seguir viviendo bastante tiempo, los tipos de interés se han ido reduciendo en los últimos meses, pero algunos economistas creen que esta tendencia va a remitir. Tiene la opción de bloquear el tipo de interés o dejar que siga fluctuando. ¿Qué haría?

    a) Sin dudas, fijar la tasa de interés

    b) Seguramente fijar la tasa de interés.

    c) Probablemente dejar que la tasa de interés siga variando.

    d) Seguro que dejar que la tasa de interés siga variando.

16. Le regalan 1.000€, elija una de las dos opciones:

    a) Una ganancia segura de 500€.

    b) Una probabilidad del 50 % de ganar 1.000€ y un 50 % de no ganar nada.

17. Le regalan 2.000€, elija una de las dos opciones:

    a) Una pérdida segura de 500€.

    b) Una probabilidad del 50 % de perder 1.000€ y un 50 % de no perder nada.

18. Un pariente le deja una herencia de 100.000€, suponga que decide invertir todo. ¿Qué opción elegiría?

    a) Una cuenta de ahorros o un fondo de inversión de deuda a corto plazo (Money Market).

    b) Un fondo de inversión de acciones y bonos.

    c) Una cartera de 15 acciones populares.

    d) Commodities como oro, plata o petróleo.

19. Si tiene que invertir 20.000€, ¿cómo distribuiría su cartera según el riesgo?

    a) 60 % en bajo riesgo, 30 % de riesgo medio y 10 % de riesgo elevado.

    b) 60 % en bajo riesgo, 30 % de riesgo medio y 10 % de riesgo elevado.

    c) 30 % en bajo riesgo, 40 % de riesgo medio y 30 % de riesgo elevado.

    d) 10 % en bajo riesgo, 40 % de riesgo medio y 50 % de riesgo elevado.

20. Un amigo de confianza, que es un experimentado geólogo, esta recaudando financiación para llevar a cabo una expedición para encontrar un yacimiento de oro con el propósito de explotarlo después. Si su amigo no logra encontrarlo, usted perderá toda su inversión. Su amigo estima que hay un 20 % de probabilidad de éxito. Si tuviese el dinero, ¿cuánto invertiría?

    a) Nada.

b) El equivalente al salario de un mes.

c) El equivalente al salario de tres meses.

d) El equivalente a seis meses de salario.

Cada pregunta puntúa de la siguiente forma:

1. a = 4; b = 2; c = 3; d = 1
2. a = 1; b = 2; c = 3; d = 4
3. a = 1; b = 2; c = 3; d = 4
4. a = 1; b = 2; c = 3
5. a = 1; b = 2; c = 3
6. a = 1; b = 2; c = 3
7. a = 2; b = 1; c = 3; d = 1
8. a = 1; b = 2; c = 3; d = 4
9. a = 1; b = 2; c = 3
10. a = 1; b = 3
11. a = 1; b = 2; c = 3
12. a = 1; b = 2; c = 3; d = 4
13. a = 1; b = 2; c = 3; d = 4
14. a = 1; b = 2; c = 3; d = 4
15. a = 1; b = 2; c = 2; d = 3
16. a = 1; b = 3
17. a = 1; b = 3
18. a = 1; b = 2; c = 3; d = 4
19. a = 1; b = 2; c = 3
20. a = 1; b = 2; c = 3; d = 4

Por último, las dimensiones del riesgo que toma en cuenta el estudio son:

1. **Garantizado vs. apuestas probables**:
   Esta dimensión requiere al usuario hacer cálculos sobre el riesgo.

2. **Elección de riesgos generales**:
   En esta dimensión se tiene en cuenta la correlación entre la elección de riesgos generales y la tolerancia al riesgo financiero. Una persona para la que el dinero es una fuente de ansiedad, estará menos predispuesta a tomar decisiones financieras arriesgas. Existe, por tanto, una relación inversa entre la ansiedad y la toma de riesgos.

3. **Elegir entre pérdida segura o ganancia segura**:
   Esta dimensión exige al usuario elegir entre distintas alternativas sin disponer de información completa.

4. **Riesgo como experiencia y conocimiento**:
   El estudio afirma que las personas que se consideran inversores experimentados o con mayor conocimiento, tienden a tomar más riesgos.

5. **Riesgo como nivel de comfort**:
   En esta dimensión se considera que existen individuos con mayor predisposición psicológica que otros a la hora de tolerar el riesgo. Hay inversores que consideran el riesgo como oportunidad, mientras que otros lo traducen como pérdida.

6. **Riesgo especulativo**:
   Esta dimensión considera que los inversores con mayor propensión a la especulación tienen una tolerancia al riesgo superior.

7. **Teoría prospectiva**:
   Esta teoría afirma que los inversores evalúan sus elecciones en términos de ganancias potenciales y pérdidas re-

lativas en relación a algún punto de referencia, que varía según la persona.

8. **Riesgo de inversión**:
   Esta dimensión toma en cuenta la combinación entre el conocimiento y el temperamento del inversor.

La relación de cada pregunta con las dimensiones del riesgo es la siguiente:

1. Garantizado vs. apuestas probables:
   2, 11, 13, 14, 15 y 20
2. Elección de riesgos generales:
   4, 13
3. Elegir entre pérdida segura o ganancia segura:
   7, 14
4. Riesgo como experiencia y conocimiento
   1, 5, 6, 8, 9, 10, 12, 15 y 18
5. Riesgo como nivel de comfort:
   1, 3, 4, 6, 7, 8, 9, 11, 13, 15 y 19
6. Riesgo especulativo:
   2, 12, 14 y 20
7. Teoría prospectiva:
   16, 17 y 19
8. Riesgo de inversión:
   5, 6, 10, 12 y 18

Para calcular el resultado, se suman todas las puntuaciones obtenidas y se establecen tres segmentos asociados a cada perfil de riesgo. El mínimo de puntuación es 20 y el máximo es 69. Así, el perfil de riesgo *bajo* corresponde a una puntuación de entre 20 y 36. El perfil *moderado* a una puntuación de entre 37 y 53. Por último, el grupo de riesgo *alto* le corresponde a las puntuaciones de entre 54 y 69.

Bibliografía

[1] P. H. de Cos, "El impacto del covid-19 en la economía española," tech. rep., CONSEJO GENERAL DE ECONOMISTAS, julio 2020.

[2] Axon Wealth Advisory Digital A.V., S.A.U., "Finizens." https://finizens.com, mayo 2021.

[3] A. Bada, "What Are the Different Types of Fintech?," *Coinspeaker*, octubre 2019.

[4] eToro (Europe) Ltd., "etoro." https://www.etoro.com/, mayo 2021.

[5] Plus500 Ltd., "Plus500." https://www.plus500.es/, mayo 2021.

[6] INDEXA CAPITAL A.V., S.A, "Indexa Capital." https://indexacapital.com/es/esp, mayo 2021.

[7] Intuit, Inc., "Mint." https://mint.intuit.com, mayo 2021.

[8] Silicon Cloud Technologies, LLC, "Portfolio visualizer." https://www.portfoliovisualizer.com/, mayo 2021.

[9] Andreas Buchen, "Portfolio performance." https://www.portfolio-performance.info/en/, mayo 2021.

[10] C. A. Juste, "Modelo de valoración de activos financieros (capm)," febrero 2017.

[11] Comisión Nacional de Mercado de Valores, "Cnmv - glosario financiero - aversión al riesgo." https://www.cnmv.es/Portal/Inversor/Glosario.aspx?id=0&term=Aversión%20al%20riesgo&idlang=1, mayo 2021.

[12] Comisión Nacional de Mercado de Valores, "Cnmv - renta variable." https://www.cnmv.es/Portal/Inversor/Renta-Variable.aspx, mayo 2021.

[13] Twelve Data Pte. Ltd., "Twelve data." https://twelvedata.com/, mayo 2021.

[14] Alpha Vantage Inc., "Alpha vantage." https://www.alphavantage.co/, mayo 2021.

[15] Verizon Media, "Yahoo finanzas." https://es.finance.yahoo.com/, mayo 2021.

[16] S. Chacon, J. Long, and other contributors, "Git." https://git-scm.com/, abril 2005.

[17] S. Chacon, J. Long, and other contributors, "Git - fundamentos de git." https://git-scm.com/book/es/v2/Inicio---Sobre-el-Control-de-Versiones-Fundamentos-de-Git, abril 2005.

[18] "GitHub: Where the world builds software," mayo 2021.

[19] G. Van Rossum and F. L. Drake Jr, *Python*. Centrum voor Wiskunde en Informatica Amsterdam, The Netherlands, 1995.

[20] JetBrains s.r.o., *PyCharm Community Edition*. JetBrains s.r.o., Na Hřebenech II 1718/10, Prague, 14000, Czech Republic, diciembre 2020.

[21] G. van Rossum, B. Warsaw, and N. Coghlan, "Style guide for Python code," PEP 8, 2001.

[22] Python Packaging Authority (PyPA), *Virtualenv*, febrero 2021.

[23] VMware, Inc., "Messaging that just works — rabbitmq." https://www.rabbitmq.com/, 2007.

[24] OASIS, "Advanced message queuing protocol (amqp) version 1.0," 2012.

[25] D. Merkel, "Docker: lightweight linux containers for consistent development and deployment," *Linux journal*, vol. 2014, no. 239, p. 2, 2014.

[26] Docker, Inc., *What is a Container? A standardized unit of software*.

[27] Docker, Inc., *Overview of Docker Compose*.

[28] O. Ben-Kiki, C. Evans, and I. döt Net, *YAML Ain't Markup Language (YAML™) Version 1.2*, octubre 2009.

[29] L. Richardson, *Beautiful Soup*, octubre 2020.

[30] MongoDB, Inc., *MongoDB*, mayo 2021.

[31] T. Haerder and A. Reuter, "Principles of transaction-oriented database recovery," *ACM Comput. Surv.*, vol. 15, p. 287–317, Dec. 1983.

[32] MongoDB, Inc., *PyMongo 3.11.4*, febrero 2021.

[33] T. Garnock-Jones, G. M. Roy, Pivotal Software, Inc, and contributors, *pika 1.2.0 documentation*, febrero 2021.

[34] A. Grönholm, *APScheduler 3.7.0 documentation*, enero 2021.

[35] The pandas development team, *Pandas v.1.2.2*, febrero 2021.

[36] C. R. Harris, K. J. Millman, S. J. van der Walt, R. Gommers, P. Virtanen, D. Cournapeau, E. Wieser, J. Taylor, S. Berg, N. J. Smith, R. Kern, M. Picus, S. Hoyer, M. H. van Kerkwijk, M. Brett, A. Haldane, J. F. del R'ıo, M. Wiebe, P. Peterson, P. G'erard-Marchant, K. Sheppard, T. Reddy, W. Weckesser, H. Abbasi, C. Gohlke, and T. E. Oliphant, *Numpy v.1.20.1*, febrero 2021.

[37] M. Grinberg and contributors, *Flask v.1.1.2*, abril 2020.

[38] N. Iarocci and contributors, *Cerberus v.1.3.3*, abril 2021.

[39] SmartBear Software, Inc., *Open Api v.3.0*, julio 2017.

[40] Postman, Inc., *Postman API Client*, mayo 2021.

[41] Django Software Foundation, *dj.choices 0.11.0*, diciembre 2020.

[42] D. Alur, D. Malks, and J. Crupi, *Core J2EE Patterns: Best Practices and Design Strategies*. USA: Prentice Hall Press, 2nd ed., 2013.

[43] D. C. E. contributors, *DBeaver Community*, septiembre 2012.

[44] Bootstrap team, *Bootstrap v.4.5.3*, octubre 2020.

[45] Chart.js contributors, *Chart.js*, abril 2021.

[46] G. Hohpe and B. Woolf, "Publish-subscribe channel - enterprise integration patterns," tech. rep., octubre 2003.

[47] T. Preston-Werner, "Versionado semántico 2.0.0," tech. rep., abril 2021.

[48] P. Eby, "Python web server gateway interface v1.0.1," PEP 3333, 2010.

[49] J. C. Deepak Alur and D. Malks, "Application service - core j2ee patterns," tech. rep., junio 2003.

[50] J. C. Deepak Alur and D. Malks, "Data access object - core j2ee patterns," tech. rep., junio 2003.

[51] Comisión Nacional de Mercado de Valores, "Cnmv - código isin." https://www.cnmv.es/portal/ANCV/CodigoISIN.aspx, mayo 2021.

[52] L. Johansson, "Part 4: Rabbitmq exchanges, routing keys and bindings - cloudamqp," *cloudamqp.com*, septiembre 2019.

[53] J. C. Deepak Alur and D. Malks, "Transfer object - core j2ee patterns," tech. rep., junio 2003.

[54] R. Aroussi, *YFinance*, marzo 2021.

[55] "Business Insider España," mayo 2021.

[56] J. Grable and R. H. Lytton, "Financial risk tolerance revisited: the development of a risk assessment instrument," *Financial Services Review*, vol. 8, no. 3, pp. 163–181, 1999.

[57] Łukasz Langa, A. Jankowski, M. Kurek, *et al.*, *Django v.3.1.4*, octubre 2016.

[58] Comisión Nacional de Mercado de Valores, "Cnmv - mercado primario." https://www.cnmv.es/Portal/Inversor/Mercado-Primario.aspx, mayo 2021.

[59] Comisión Nacional de Mercado de Valores, "Cnmv - mercado secundario." https://www.cnmv.es/Portal/Inversor/Mercado-Secundario.aspx, mayo 2021.

[60] Comisión Nacional de Mercado de Valores, "Cnmv - glosario financiero - ibex 35." https://www.cnmv.es/Portal/Inversor/Glosario.aspx?id=0&term=Ibex%2035&idlang=1, mayo 2021.

[61] R. V. Burguillo, "Diferencia entre mercado primario y secundario," *Economipedia.com*, febrero 2016.

[62] R. V. Burguillo, "Mercado primario," *Economipedia.com*, enero 2016.

[63] R. V. Burguillo, "Mercado secundario," *Economipedia.com*, enero 2016.

[64] A. S. Arias, "Índice bursátil - qué es, definición y concepto," *Economipedia.com*, febrero 2012.

[65] B. H. Blázquez, *Compendio bursátil*. Ediciones Díaz de Santos, 2014.

[66] Real Python Team, "Python virtual environments: A primer," *Real Python*, enero 2018.

[67] L. Johansson, "Part 1: Rabbitmq for beginners - what is rabbitmq?," *cloudamqp.com*, septiembre 2019.

[68] Python Software Foundation, "Python software foundation." https://www.python.org/psf/, mayo 2021.

[69] J. C. Deepak Alur and D. Malks, "Business object - core j2ee patterns," tech. rep., junio 2003.

[70] M. Fowler and J. Lewis, "Microservices a definition of this new architectural term," *martinfowler.com*, marzo 2014.

[71] C. R. Bacon, *Practical portfolio performance : measurement and attribution LK - https://ucm.on.worldcat.org/oclc/646771663*. Wiley finance series, Chichester, England ;: Wiley, 2nd ed. nv ed., 2008.

[72] R. C. Marston, *Portfolio design : a modern approach to asset allocation LK - https://ucm.on.worldcat.org/oclc/708037290*. [Wiley finance], Hoboken, NJ: Wiley, nv - 1 onl ed., 2011.

Gerardo Parra Rossignoli
Junio 2021
Ult. actualización 15 de junio de 2021
LaTeX lic. LPPL & powered by TEFLON CC-ZERO